36.1.1. Understanding Inheritance

05:53

Inheritance is a way by which a class can inherit what is already there in another existing class.

We can classify inheritance into two kinds:

1. **Implementation Inheritance** (or code inheritance)
2. **Interface Inheritance** (or inheritance of behaviour or contract)

Here we will understand *implementation inheritance* and later learn about *interface inheritance* after we learn *interfaces*.

In Java, a class can **inherit** implementation from **only one class**, it is also referred as **single-inheritance**. (Other languages like c++ support multiple inheritance of implementation)

In Java, we use `extends` keyword when we want a class to inherit (extend) from another class. For example:

```
class A {
    ...
}
class B extends A {
    ...
}
```

In the above code B extends A. B is called the subclass of A. A is called the super class of B.

Note: Every class in Java automatically extends the root class Object if it does not explicitly extends another class. Which means in our above example, class A is a subclass of Object and Object is the superclass of A.

Using the above concepts select all the correct statements from the below code:

```
class W { // statement 1
}
class X { // statement 2
    ...
```

- [ ] Statement 1 is wrong. Since every class in Java extends `Object` class, statement 1 should have been

  ```
  class W extends Object {
  ```

- [ ] As per the class declaration in statement 3, `Y` is the superclass of `X`.
- [ ] As per statement 2, the class declaration statement of `X` states that there is no superclass for `X`.
- [ ] Statement 4, which states that class `Z1` is the subclass of both `W` and `Y` is correct.
- [x] As per statement 5, `Z2` is the subclass of `Y`, `X` and `Object`.

< Prev Reset Submit Next >

36.1.2. Understanding implementation Inheritance usage

54:39

In implementation inheritance the subclass inherits the implementation facilitating code reuse. For example:

```
class A {
    public int aValue = 1;
    public int getAValue() {
        return aValue;
    }
}
class B extends A {
    public int bValue = 2;
    public int getBValue() {
        return bValue;
    }
}
```

In the above code, class `B` also contains the inherited field `aValue` and the inherited method `getAValue()`.

Observe the given code in editor and complete the incomplete code:

- It demonstrates inheritance with two classes: **Person** and **Student**.
- The **Person** class has instance variables **name** and **age**, along with **getInfo()** and **displayInfo()** methods.
- In the **getInfo()** method of the **Student** class, the user is prompted to enter the name, age, and roll number of a student. The input is stored in the respective variables using the **super** keyword to access the parent class's **getInfo()** method.
- The **displayInfo()** method of the **Student** class overrides the parent class's method and displays the student's information along with the roll number.
- In the **main()** method, a **Student** object is created, and the **getInfo()** and **displayInfo()** methods

Sample Test Cases +

Explorer Inheritan... Submit Plots

```
20      }
21  }
22
23  //Complete the code
24  class Student extends Person {
25      private int rollNumber;
26      public void getInfo(){
27          super.getInfo();
28          Scanner scanner =new Scanner(System.in);
29          System.out.print("Enter the roll number: ");
30          rollNumber=scanner.nextInt();
31      }
32      public void displayInfo(){
33          super.displayInfo();
34          System.out.println("Roll Number: "+rollNumber);
35      }
36
37
38
39
40  }
41
42
43  public class InheritanceExample {
44      public static void main(String[] args) {
45          Student student = new Student();
46          student.getInfo();
47          student.displayInfo();
48      }
49  }
50
```

Terminal Test cases

< Prev Reset Submit Next >

36.1.3. Understanding implementation inheritance usage

37:16

In implementation inheritance, when the subclass inherits from a superclass (also callled base class), the subclass instance can be referred by the base class reference. For example:

```
class A {
    public int aValue = 1;
    public int getAValue() {
        return aValue;
    }
}

class B extends A {
    public int bValue = 2;
    public int getBValue() {
        return bValue;
    }
}
Then the below statement is valid:
A a = new B();
```

Note: In the above code, we can access all members of class `A` via the reference `a` even though the actual object created in memory using the `new` keyword is an instance of class `B`.

According to the above code, the statement `a.getBValue()` will result in compilation error. This is because reference `a` which is of type class `A` does not know about the members declared in its subclass `B`.

See and observe the code in the editor and complete it:

- The **A** class has an instance variable **aValue**, along with a **getAValue()** method.
- The **B** class must be inherited from **A** and will have an additional instance variable **bValue** and a **getBValue()** method.
- In the **main()** method, the user is prompted to enter the value for **A** and **B** using the **Scanner**

Sample Test Cases +

Explorer

Inheritan...

Submit

Plots

```
1  package q35966;
2  import java.util.Scanner;
3
4  class A {
5      public int aValue;
6
7      public int getAValue() {
8          return aValue;
9      }
10 }
11 class B extends A{
12     public int bValue;
13     public int getBValue(){
14         return bValue;
15     }
16 }
17 //define a class B that inherits from A
18 public class InheritanceExample2 {
19     public static void main(String[] args) {
20         Scanner scanner = new Scanner(System.in);
21         //define a reference variable of class B
22         B b=new B();
23         System.out.print("Enter the value for A: ");
24         int aValue = scanner.nextInt();
25         b.aValue = aValue;
26
27         System.out.print("Enter the value for B: ");
28         int bValue = scanner.nextInt();
29         b.bValue = bValue;
30
31         System.out.println("b.getAValue(): " + b.getAValue());
```

Terminal Test cases

36.1.4. Write a Java program to implement Single Inheritance

19:14

Write a Java program to illustrate the **single inheritance** concept.

Create a class `Marks`

- contains the data members **id** of **int** data type, **javaMarks**, **cMarks** and **cppMarks** of **float** data type
- write a method **setMarks()** to initialize the data members
- write a method **displayMarks()** which will display the given data

Create another class `Result` which is derived from the class `Marks`

- contains the data members **total** and **avg** of **float** data type
- write a method **compute()** to find total and average of the given marks
- write a method **showResult()** which will display the total and avg marks

Write a class `SingleInheritanceDemo` with **main()** method it receives four arguments as **id**, **javaMarks**, **cMarks** and **cppMarks**.

Create object only to the class `Result` to access the methods.

If the input is given as command line arguments to the **main()** as **"101", "45.50", "67.75", "72.25"** then the program should print the output as:

```
Id : 101
Java marks : 45.5
C marks : 67.75
Cpp marks : 72.25
Total : 185.5
Avg : 61.833332
```

Note: While computing the total marks, add the marks in the following order only **javaMarks, cMarks**

Sample Test Cases +

Explorer SingleInh... Submit Plots

```
1  package q11263;
2  class Marks{
3      int id;
4      float javaMarks;
5      float cMarks;
6      float cppMarks;
7      void setMarks(int id,float javaMarks,float cMarks,float
       cppMarks){
8          this.id=id;
9          this.javaMarks=javaMarks;
10         this.cMarks=cMarks;
11         this.cppMarks=cppMarks;
12     }
13     void displayMarks(){
14         System.out.println("Id : "+id);
15         System.out.println("Java marks : "+javaMarks);
16         System.out.println("C marks : "+cMarks);
17         System.out.println("Cpp marks : "+cppMarks);
18     }
19 }
20 class Result extends Marks{
21     float total;
22     float avg;
23     void compute(){
24         total= javaMarks+cMarks+cppMarks;
25         avg=total/3;
26     }
27     void showResult(){
28         System.out.println("Total : "+total);
29         System.out.println("Avg : "+avg);
30     }
```

Terminal Test cases

Prev Reset Submit Next

36.1.4. Write a Java program to Implement Single Inheritance

19:14

Write a Java program to illustrate the **single inheritance** concept.

Create a class `Marks`

- contains the data members **id** of **int** data type, **javaMarks**, **cMarks** and **cppMarks** of **float** data type
- write a method **setMarks()** to initialize the data members
- write a method **displayMarks()** which will display the given data

Create another class `Result` which is derived from the class `Marks`

- contains the data members **total** and **avg** of **float** data type
- write a method **compute()** to find total and average of the given marks
- write a method **showResult()** which will display the total and avg marks

Write a class `SingleInheritanceDemo` with **main()** method it receives four arguments as **id**, **javaMarks**, **cMarks** and **cppMarks**.

Create object only to the class `Result` to access the methods.

If the input is given as command line arguments to the **main()** as **"101", "45.50", "67.75", "72.25"** then the program should print the output as:

```
Id : 101
Java marks : 45.5
C marks : 67.75
Cpp marks : 72.25
Total : 185.5
Avg : 61.833332
```

Note: While computing the total marks, add the marks in the following order only **javaMarks**, **cMarks**

Sample Test Cases +

Explorer

SingleInh...

Submit

Plots

```
14      System.out.println("Id : "+id);
15      System.out.println("Java marks : "+javaMarks);
16      System.out.println("C marks : "+cMarks);
17      System.out.println("Cpp marks : "+cppMarks);
18  }
19  }
20  class Result extends Marks{
21  float total;
22  float avg;
23  void compute(){
24  total= javaMarks+cMarks+cppMarks;
25  avg=total/3;
26  }
27  void showResult(){
28  System.out.println("Total : "+total);
29  System.out.println("Avg : "+avg);
30  }
31  }
32  class SingleInheritanceDemo{
33  public static void main(String[] args){
34  int id=Integer.parseInt(args[0]);
35  float javaMarks=Float.parseFloat(args[1]);
36  float cMarks=Float.parseFloat(args[2]);
37  float cppMarks=Float.parseFloat(args[3]);
38  Result obj=new Result();
39  obj.setMarks(id,javaMarks,cMarks,cppMarks);
40  obj.displayMarks();
41  obj.compute();
42  obj.showResult();
43  }
44  }
```

Terminal Test cases

< Prev Reset Submit Next >

36.1.5. Write a Java program to implement Multilevel Inheritance

13:15

Write a Java program to illustrate the **multilevel inheritance** concept.

Create a class `Student`
- contains the data members **id** of **int** data type and **name** of **string** type
- write a method **setData()** to initialize the data members
- write a method **displayData()** which will display the given **id** and **name**

Create a class `Marks` which is derived from the class `Student`
- contains the data members **javaMarks**, **cMarks** and **cppMarks** of **float** data type
- write a method **setMarks()** to initialize the data members
- write a method **displayMarks()** which will display the given data

Create another class `Result` which is derived from the class `Marks`
- contains the data members **total** and **avg** of **float** data type
- write a method **compute()** to find total and average of the given marks
- write a method **showResult()** which will display the total and avg marks

Write a class `MultilevelInheritanceDemo` with the **main()** method which will receive five arguments as **id**, **name**, **javaMarks**, **cMarks** and **cppMarks**.

Create object only to the class `Result` to access the methods.

If the input is given as command line arguments to the **main()** as **"99", "Lakshmi", "55.5", "78.5", "72"** then the program should print the output as:

```
Id : 99
Name : Lakshmi
Java marks : 55.5
C marks : 78.5
Cpp marks : 72.0
Total : 206.0
```

Sample Test Cases +

Explorer

Multilevel...

Submit

Plots

```
1  package q11264;
2  class Student{
3      int id;
4      String name;
5      void setData(int id,String name){
6          this.id=id;
7          this.name=name;
8      }
9      void displayData(){
10         System.out.println("Id : "+id);
11         System.out.println("Name : "+name);
12     }
13 }
14 class Marks extends Student{
15     float javaMarks;
16     float cMarks;
17     float cppMarks;
18     void setMarks(float javaMarks,float cMarks,float cppMarks){
19         this.javaMarks=javaMarks;
20         this.cMarks=cMarks;
21         this.cppMarks=cppMarks;
22     }
23     void displayMarks(){
24         System.out.println("Java marks : "+javaMarks);
25         System.out.println("C marks : "+cMarks);
26         System.out.println("Cpp marks : "+cppMarks);
27     }
28 }
29 class Result extends Marks{
30     float total;
31     float avg;
```

Terminal Test cases

Prev Reset Submit Next

36.1.5. Write a Java program to implement Multilevel Inheritance

13:15

Write a Java program to illustrate the **multilevel inheritance** concept.

Create a class `Student`
- contains the data members **id** of **int** data type and **name** of **string** type
- write a method **setData()** to initialize the data members
- write a method **displayData()** which will display the given **id** and **name**

Create a class `Marks` which is derived from the class `Student`
- contains the data members **javaMarks**, **cMarks** and **cppMarks** of **float** data type
- write a method **setMarks()** to initialize the data members
- write a method **displayMarks()** which will display the given data

Create another class `Result` which is derived from the class `Marks`
- contains the data members **total** and **avg** of **float** data type
- write a method **compute()** to find total and average of the given marks
- write a method **showResult()** which will display the total and avg marks

Write a class `MultilevelInheritanceDemo` with the **main()** method which will receive five arguments as **id**, **name**, **javaMarks**, **cMarks** and **cppMarks**.

Create object only to the class `Result` to access the methods.

If the input is given as command line arguments to the **main()** as **"99", "Lakshmi", "55.5", "78.5", "72"** then the program should print the output as:

```
Id : 99
Name : Lakshmi
Java marks : 55.5
C marks : 78.5
Cpp marks : 72.0
Total : 206.0
```

Sample Test Cases +

Explorer

Multilevel...

Submit

Plots

```
26          System.out.println("Cpp marks : "+cppMarks);
27      }
28  }
29  class Result extends Marks{
30      float total;
31      float avg;
32      void compute(){
33          total=javaMarks+cMarks+cppMarks;
34          avg=total/3;
35      }
36      void showResult(){
37          System.out.println("Total : "+total);
38          System.out.println("Avg : "+avg);
39      }
40  }
41  class MultilevelInheritanceDemo{
42      public static void main(String args[]){
43          int id=Integer.parseInt(args[0]);
44          String name=args[1];
45          float javaMarks=Float.parseFloat(args[2]);
46          float cMarks=Float.parseFloat(args[3]);
47          float cppMarks=Float.parseFloat(args[4]);
48          Result stu=new Result();
49          stu.setData(id,name);
50          stu.displayData();
51          stu.setMarks(javaMarks,cMarks,cppMarks);
52          stu.displayMarks();
53          stu.compute();
54          stu.showResult();
55      }
56  }
```

Terminal Test cases

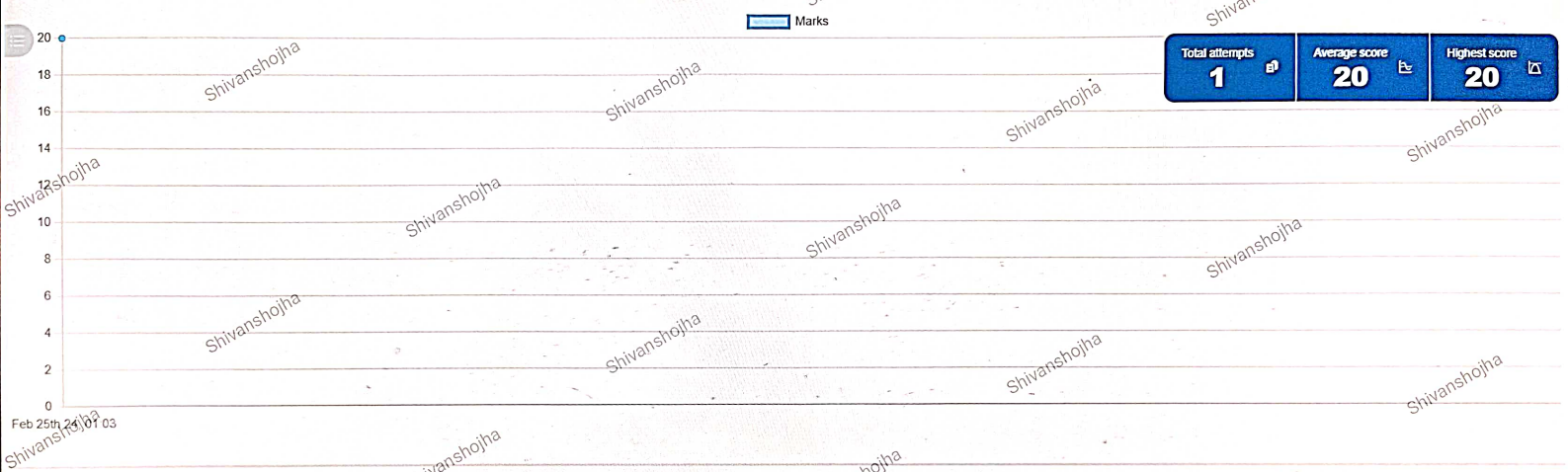
Prev Reset Submit Next

Home Learn Anywhere 12216507.st@lpu.in Support Logout

Java - Inheritance - I

Try Again

2 hours 20 marks(s)



Total attempts	Average score	Highest score
1	20	20

Attempt	Test date	Results	Status	Actions
1	February 25th 2024, 1:03:15 am	20/20	Passed	Details

< Prev Next >

37.1.1. Understanding Method Overloading

05:05

Method overloading means the ability to have multiple methods with same name, which vary in their parameters. For example:

```
public void concatenate(String text, int num) {
    return text + num;
}
public void concatenate(String text, boolean flag) {
    return text + flag;
}
public void concatenate(String text, char ch) {
    return text + ch;
}
```

In the above code, `concatenate` method is overloaded **three** times.

Note: In the above example, the variation in the parameters list can be by their count or type or both.

Select all the correct statements given below regarding methods present in the String class.

[**Hint:** You can explore the methods present in the String class by clicking on String and scrolling down until you reach a section heading named **Method Summary**. In that you will see table containing a list of all methods in the String class.]

- [x] `charAt` method is not overloaded.
- [x] `indexOf` method is overloaded **4 times**.
- [x] The `String` constructor is overloaded **15 times** .
- [ ] `valueOf` method is overloaded **8 times**.

37.1.2. Write a Java program to Implement Method overloading

12:14

Write a Java program with a class name `Addition` with the methods `add(int, int)`, `add(int, float)`, `add(float, float)` and `add(float, double, double)` to add values of different argument types.

Write the **main(String[])** method within the class and assume that it will always receive a total of **6** command line arguments at least, such that the first **2** are **int**, next **2** are **float** and the last **2** are of type **double**.

If the **main()** is provided with arguments : **1, 2, 1.5f, 2.5f, 1.0, 2.0** then the program should print the output as:

```
Sum of 1 and 2 : 3
Sum of 1.5 and 2.5 : 4.0
Sum of 2 and 2.5 : 4.5
Sum of 1.5, 1.0 and 2.0 : 4.5
```

Note: Please don't change the package name.

Sample Test Cases +

Explorer Addition.j... Submit Plots

```
1   package q11266;
2   class Addition{
3       int add(int a,int b){
4           return a+b;
5       }
6       float add(int a, float b){
7           return a+b;
8       }
9       float add(float a, float b){
10          return a+b;
11      }
12      double add(float a, double b, double c){
13          return a+b+c;
14      }
15      public static void main(String[] args){
16          int a=Integer.parseInt(args[0]);
17          int b=Integer.parseInt(args[1]);
18          float c=Float.parseFloat(args[2]);
19          float d=Float.parseFloat(args[3]);
20          double e=Double.parseDouble(args[4]);
21          double f=Double.parseDouble(args[5]);
22          Addition obj=new Addition();
23          System.out.println("Sum of "+a+" and "+b+" :
    "+obj.add(a,b));
24
25          System.out.println("Sum of "+c+" and "+d+" :
    "+obj.add(c,d));
26          System.out.println("Sum of "+b+" and "+d+" :
    "+obj.add(b,d));
27          System.out.println("Sum of "+c+", "+e+" and "+f+" :
    "+obj.add(c,e,f));
```

Terminal Test cases

Prev Reset Submit Next

37.1.3. Write a Java program to Implement Constructor overloading

17:26

Write a class `Box` which contains the data members **width**, **height** and **depth** all of type **double**.

Write the implementation for the below **3** *overloaded constructors* in the class `Box` :

- **Box()** - default constructor which initializes all the members with **-1**
- **Box(length)** - parameterized constructor with one argument and initialize all the members with the value in **length**

the members with the corresponding arguments

- **Box(width, height, depth)** - parameterized constructor with three arguments and initialize

Write a method `public double volume()` in the class `Box` to find out the **volume** of the given box.

Write the **main** method within the `Box` class and assume that it will receive either **zero** arguments, or **one** argument or **three** arguments.

For example, if the **main()** method is passed **zero** arguments then the program should print the output as:

```
Volume of Box() is : -1.0
```

Similarly, if the **main()** method is passed **one** argument : **2.34**, then the program should print the output as:

```
Volume of Box(2.34) is : 12.812903999999998
```

then the program should print the output as: Likewise, if the **main()** method is passed **three** arguments : **2.34, 3.45, 1.59**, then the program should print the output as:

```
Volume of Box(2.34, 3.45, 1.59) is : 12.836070000000001
```

Sample Test Cases +

Explorer

Box.java

Submit

```
1  package q11267;
2  class Box{
3      double width;
4      double height;
5      double depth;
6      Box(){
7          width=-1;
8          height=-1;
9          depth=-1;
10     }
11     Box(double length){
12         width=length;
13         height=length;
14         depth=length;
15     }
16     Box(double width,double height,double depth){
17         this.width=width;
18         this.height=height;
19         this.depth=depth;
20     }
21     public double volume(){
22         return width*height*depth;
23     }
24     public static void main(String[] args){
25         if(args.length==0)
26         {
27             Box obj=new Box();
28             System.out.println("Volume of Box() is : "+obj.volume());
29         }
30         else if(args.length==1){
31             double length=Double.parseDouble(args[0]);
```

Terminal Test cases

< Prev Reset Submit Next >

37.1.3. Write a Java program to Implement Constructor overloading

17:26

Write a class `Box` which contains the data members **width**, **height** and **depth** all of type **double**.

Write the implementation for the below **3** *overloaded constructors* in the class `Box` :

- **Box()** - default constructor which initializes all the members with **-1**
- **Box(length)** - parameterized constructor with one argument and initialize all the members with the value in **length**
- **Box(width, height, depth)** - parameterized constructor with three arguments and initialize the members with the corresponding arguments

Write a method `public double volume()` in the class `Box` to find out the **volume** of the given box.

Write the **main** method within the `Box` class and assume that it will receive either **zero** arguments, or **one** argument or **three** arguments.

For example, if the **main()** method is passed **zero** arguments then the program should print the output as:

```
Volume of Box() is : -1.0
```

Similarly, if the **main()** method is passed **one** argument : **2.34**, then the program should print the output as:

```
Volume of Box(2.34) is : 12.812903999999998
```

then the program should print the output as: Likewise, if the **main()** method is passed **three** arguments : **2.34, 3.45, 1.59**, then the program should print the output as:

```
Volume of Box(2.34, 3.45, 1.59) is : 12.836070000000001
```

Sample Test Cases +

Explorer

Box.java

Submit

```
15      }
16      Box(double width,double height,double depth){
17          this.width=width;
18          this.height=height;
19          this.depth=depth;
20      }
21      public double volume(){
22          return width*height*depth;
23      }
24      public static void main(String[] args){
25          if(args.length==0)
26          {
27              Box obj=new Box();
28              System.out.println("Volume of Box() is : "+obj.volume());
29          }
30          else if(args.length==1){
31              double length=Double.parseDouble(args[0]);
32              Box obj=new Box(length);
33              System.out.println("Volume of Box("+length+") is :
    "+obj.volume());
34          }
35          else{
36              double width=Double.parseDouble(args[0]);
37              double height=Double.parseDouble(args[1]);
38              double depth=Double.parseDouble(args[2]);
39              Box obj=new Box(width,height,depth);
40              System.out.println("Volume of Box("+width+", "+height+",
    "+depth+") is : "+obj.volume());
41          }
42      }
43  }
```

Terminal Test cases

< Prev Reset Submit Next >

37.1.4. Write a Java program to implement Method overloading

10:17

Write a Java program with a class name `OverloadArea` with overload methods `area(float)` and `area(float, float)` to find area of **square** and **rectangle**.

Write the **main** method within the class and assume that it will receive a total of **2** command line arguments of type **float**.

If the **main()** is provided with arguments : **1.34, 1.98** then the program should print the output as:

```
Area of square for side in meters 1.34 : 1.7956
Area of rectangle for length and breadth in meters 1.34, 1.98 : 2.6532001
```

Note: Please don't change the package name.

Sample Test Cases +

Explorer

Overload...

Submit

Plots

```
1  package q11268;
2  public class OverloadArea {
3      // Write the overload methods
4      float area(float side){
5          return side*side;
6      }
7      float area(float len,float bre){
8          return len*bre;
9      }
10     public static void main (String[] args) {
11         // Write the code
12         OverloadArea obj=new OverloadArea();
13         float f1=Float.parseFloat(args[0]);
14         System.out.println("Area of square for side in meters " +
           f1 +" : " +obj.area(f1) );
15         float f2=Float.parseFloat(args[1]);
16         System.out.println("Area of rectangle for length and
           breadth in meters " +f1 +", " +f2 +" : " +obj.area(f1,f2) );
           // Fill in the missing code
17     }
18 }
19
```

Terminal Test cases

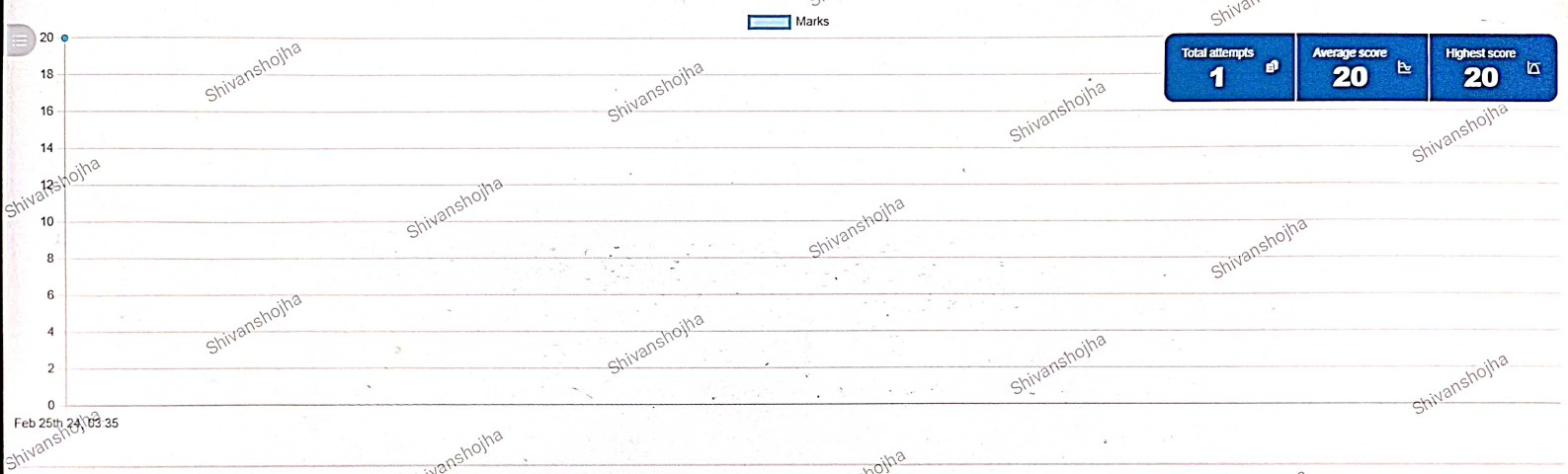
Prev Reset Submit Next

Home Learn Anywhere 12216507.st@lpu.in Support Logout

Java - Inheritance - II

Try Again

2 hours 20 marks(s)



Total attempts	Average score	Highest score
1	20	20

Attempt	Test date	Results	Status	Actions
1	February 25th 2024, 3:35:35 am	20/20	Passed	Details

< Prev Next >

38.1.1. Understanding Overriding

02:48

In implementation of inheritance, when a class B inherits from a class A, the subclass B can modify the implementation present in its superclass A. For example:

```
class A {
    public int aValue = 2;
    public int getAValue() {
        return aValue;
    }
}

class B extends A {
    public int bValue = 3;
    public int getBValue() {
        return bValue;
    }
    public int getAValue() { // this method overrides the implementation in class A
        return 2 * aValue; // returning  double of value stored in aValue
    }
}
Then the below will print 4 and not 2:
B b = new B();
System.out.println(b.getAValue());
```

Note: In the above code, the method `getAValue()` in `class B` is overriding the method with same name present in its superclass `A`.

Whenever a method is overridden in the subclass, the new method implementation will be called when the method is invoked on the instance of subclass.

Observe and understand the code given in the editor and complete it:

- The **A** class has an instance variable **aValue**, along with a **getAValue()** method.

Sample Test Cases

Explorer

Overridin...

Submit

Plots

```
1  import java.util.Scanner;
2
3  class A {
4      public int aValue;
5
6      public int getAValue() {
7          return aValue;
8      }
9  }
10
11 class B extends A {
12     public int bValue;
13     public int getBValue(){
14         return bValue;
15     }
16     public int getAValue(){
17         return 2*(super.aValue);
18     }
19
20
21 }
22
23 public class OverridingExample {
24     public static void main(String[] args) {
25         Scanner scanner = new Scanner(System.in);
26
27         B b = new B();
28
29         System.out.print("Enter the value for A: ");
30         int aValue = scanner.nextInt();
31         b.aValue = aValue;
```

Terminal Test cases

Prev Reset Submit Next

38.1.2. Difference between overloading and overriding

01:00

Method overloading is a feature which allows multiple methods with same name and different parameters in the same class.

Method overriding is a feature where we specialize (or modify) a method behaviour which is already present in the superclass. This takes places only when we have a class extending another class.

Select all the correct statements from the below code:

```
class A {
    public void printMe(int number) { // statement 1
        System.out.println(number);
    }
    public void printMe(boolean flag) { // statement 2
        System.out.println(flag);
    }
    public void printMe(int number, boolean flag) { // statement 3
        System.out.println(number + " : " + flag);
    }
}
class B extends A {
    public void printMe(int number) { // statement 4
        System.out.println("The double of " + number + " is : " + (number * 2));
    }
}
```

- [x] Statements 1, 2 and 3 in class A contain overloaded versions of method `printMe`.
- [ ] Statements 1, 2 and 3 in class A contain overridden versions of method `printMe`.
- [ ] The method in statement 4, overrides the method declared in statement 2.
- [x] The method in statement 4, overrides the method declared in statement 1.
- [ ] The method declaration in statement 3 has more parameters than the methods declared in statements 1 and 2, hence it will not be considered as a overloaded version of `printMe`.

38.1.3. Write a Java program to achieve concept of Method Overriding

07:26

Assume there is a class called `Bank` with method `calculateInterest(float principal, int time)`.

Create sub-classes of `Bank` with names `SBI`, `ICICI` and `AXIS` and override the `calculateInterest(float principal, int time)` method.

Create a constant of type `float` called `INTEREST_RATE` in classes `SBI`, `ICICI` and `AXIS` with values `10.8`, `11.6` and `12.3` respectively.

Use the formula **(principal * INTEREST_RATE * time) / 100** to calculate the **interest** for given **principal** and **time** and return the value as `float` in the **overriden method**.

For example, if the **two** arguments passed to the **main** method are 1000 and 5, (principal and time) below is the expected output:

```
SBI rate of interest = 540.0
ICICI rate of interest = 580.0
AXIS rate of interest = 615.0
```

Note: Please don't change the package name.

Sample Test Cases +

Explorer TestOverr... Submit Plots

```
1  package q11271;
2  class Bank {
3      float calculateInterest(float principal, int time){
4          return 0;
5      }
6  }
7  class SBI extends Bank {
8      private static final float INTEREST_RATE = 10.8f;
9      public float calculateInterest(float principal,int time){
10         return (principal*INTEREST_RATE*time)/100;
11     }
12 }
13 class ICICI extends Bank {
14     private static final float INTEREST_RATE = 11.6f;
15     public float calculateInterest(float principal,int time){
16         return (principal*INTEREST_RATE*time)/100;
17     }
18 }
19
20 class AXIS extends Bank {
21     private static final float INTEREST_RATE = 12.3f;
22     public float calculateInterest(float principal,int time){
23         return (principal*INTEREST_RATE*time)/100;
24     }
25 }
26
27 public class TestOverriding {
28     public static void main(String[] args) {
29         Bank sbiBank = new SBI();
30         Bank iciciBank = new ICICI();
31         Bank axisBank = new AXIS();
```

Terminal Test cases

< Prev Reset Submit Next >

38.1.3. Write a Java program to achieve concept of Method Overriding

07:26

Assume there is a class called `Bank` with method `calculateInterest(float principal, int time)`.

Create sub-classes of `Bank` with names `SBI`, `ICICI` and `AXIS` and override the `calculateInterest(float principal, int time)` method.

Create a constant of type `float` called `INTEREST_RATE` in classes `SBI`, `ICICI` and `AXIS` with values `10.8`, `11.6` and `12.3` respectively.

Use the formula **(principal * INTEREST_RATE * time) / 100** to calculate the **interest** for given **principal** and **time** and return the value as `float` in the **overriden method**.

For example, if the **two** arguments passed to the **main** method are 1000 and 5, (principal and time) below is the expected output:

```
SBI rate of interest = 540.0
ICICI rate of interest = 580.0
AXIS rate of interest = 615.0
```

Note: Please don't change the package name.

Sample Test Cases +

Explorer TestOverr... Submit Plots

```
11      }
12  }
13  class ICICI extends Bank {
14      private static final float INTEREST_RATE = 11.6f;
15      public float calculateInterest(float principal,int time){
16          return (principal*INTEREST_RATE*time)/100;
17      }
18  }
19
20  class AXIS extends Bank {
21      private static final float INTEREST_RATE = 12.3f;
22      public float calculateInterest(float principal,int time){
23          return (principal*INTEREST_RATE*time)/100;
24      }
25  }
26
27  public class TestOverriding {
28      public static void main(String[] args) {
29          Bank sbiBank = new SBI();
30          Bank iciciBank = new ICICI();
31          Bank axisBank = new AXIS();
32          float principal = Float.parseFloat(args[0]);
33          int time = Integer.parseInt(args[1]);
34          System.out.println("SBI rate of interest = "
            +sbiBank.calculateInterest(principal,time) );
35          System.out.println("ICICI rate of interest = "+
            iciciBank.calculateInterest(principal,time));
36          System.out.println("AXIS rate of interest = " +
            axisBank.calculateInterest(principal,time));
37      }
38  }
```

Terminal Test cases

Java - Method Overriding - I

Try Again

2 hours 15 marks(s)

Total attempts	Average score	Highest score
1	12.75	12.75

Attempt	Test date	Results	Status	Actions
1	February 28th 2024, 12:44:51 am	12.75/15	Passed	Details

39.1.1. Understanding super keyword

18:03

The `super` keyword in **java** is used to refer/access either a member field, method or a constructor present in the super class hierarchy of the current class where the `super` keyword is used.

The **super** keyword can be used to :

- access the member fields of parent class when both parent and child class have member fields with same name
- explicitly call the default or parameterized constructors of parent class
- access the method of parent class when child class has overridden that method

When both the child and parent class contain variables with same names, we can access the member field of parent class inside the child class by using the syntax:

```
super.variableName
```

When an **instance** of subclass is created, the **new** keyword invokes the **constructor** of child class. This invocation of constructor in the subclass **implicitly** invokes the constructor of the **parent** class, if we do not explicitly write code to invoke the constructor of super class using **super**.

Hence, we should always remember that when we create an object of child class, the **parent** class constructor is executed first and then the **child** class constructor is executed.

Supposing we do not write code to call the super class constructor in the child class constructor, then the compiler implicitly adds `super()` (this invokes the **no argument constructor** of parent class) as the first statement in the constructor of **child** class. If the parent class does not have a default constructor (constructor with no parameters), then the programmer should explicitly call the parameterized constructor else the compiler will flag and error saying there is no default constructor in the super class.

Sample Test Cases +

Explorer

SuperKey...

Submit

```
class SuperClass {
    int num;
    public SuperClass(int value) {
        num = value;
    }
    public void printHello() {
        System.out.println("Hello from SuperClass");
    }
}

class SubClass extends SuperClass {
    //Write your code here...
    SubClass(int value){
        super(value);
    }
public void printHello(){
    System.out.println("SuperClass number = "+num);
    System.out.println("SubClass number = "+15);
    super.printHello();
    System.out.println("Hello from SubClass");

}

}

public class SuperKeyword {
    public static void main(String[] args) {
        SubClass obj = new SubClass(10);
        obj.printHello();
    }
}
```

Terminal Test cases

Plots

Prev Reset Submit Next

39.1.2. Write a Java program to illustrate super keyword

03:31

Write a Java program to illustrate the usage of **super** keyword.

Create a class called `Animal` with the below members:

- a constructor which prints **Animal is created**
- a method called **eat()** which will print **Eating something** and returns nothing.

Create another class called `Dog` which is derived from the class `Animal`, and has the below members:

- a constructor which calls **super()** and then prints **Dog is created**
- a method **eat()** which will print **Eating bread** and returns nothing
- a method **bark()** which will print **Barking** and returns nothing
- a method **work()** which will call **eat()** of the **superclass** first and then the **eat()** method in the current class, followed by the **bark()** method in the current class.

Write a class `ExampleOnSuper` with the **main()** method, create an object to `Dog` which calls the method **work()**.

Note: Please don't change the package name.

Sample Test Cases +

Explorer

Example...

Submit

Plots

```
1  package q11273;
2  class Animal {
3      public Animal() {
4          System.out.println("Animal is created");
5      }
6      void eat() {
7          System.out.println("Eating something");
8      }
9  }
10 class Dog extends Animal {
11     public Dog() {
12         super();
13         System.out.println("Dog is created");
14     }
15     void eat() {
16         System.out.println("Eating bread");
17     }
18     void bark() {
19         System.out.println("Barking");
20     }
21     void work() {
22         super.eat();
23         eat();
24         bark();
25     }
26 }
27 public class ExampleOnSuper {
28     public static void main(String args[]) {
29         Dog d = new Dog();
30         d.work();
31     }
```

Terminal Test cases

< Prev Reset Submit Next >

39.1.3. Write a Java program to Access the Class members using super Keyword

12:05

Write a Java program to access the class members using **super** Keyword.

Create a class called `SuperClass` with the below members:

- declare two member fields **value1** and **value2** of type **int**
- a parameterized constructor with **two** arguments, which assigns two arguments to the members respectively
- a method called **show()** which will print **This is super class show() method** as well as the value of **value1**.

Create another class called `SubClass` which is derived from the class `SuperClass`, and has the below members:

- declare two member fields **value3** and **value4** of type **int**
- a parameterized constructor with **four** arguments, which assigns the first two arguments with **SuperClass** members and next two values with **SubClass** members
- a method called **show()** which
  1. will print **This is sub class show() method**
  2. will call **show()** of **SuperClass**
  3. will print **value2** from **SuperClass**
  4. will print **value3** of **SubClass**
  5. will print **value4** of **SubClass**

Write a class `AccessUsingSuper` with the **main()** method, create an object to `SubClass` which calls the method **show()**.

For example, if the input is given as [10, 20, 30, 40] then the output should be:

```
This is sub class show() method
This is super class show() method
value1 = 10
```

Sample Test Cases +

Explorer

AccessU...

Submit

Plots

```
1  package q11274;
2  class SuperClass {
3      int value1, value2;
4      // Write the code
5      SuperClass(int value1,int value2){
6          this.value1=value1;
7          this.value2=value2;
8      }
9      void show(){
10         System.out.println("This is super class show() method");
11         System.out.println("value1 = "+value1);
12     }
13 }
14
15 class SubClass extends SuperClass {
16     int value3, value4;
17     SubClass(int value1,int value2,int value3,int value4){
18         super(value1,value2);
19         this.value3=value3;
20         this.value4=value4;
21     }
22     void show(){
23         System.out.println("This is sub class show() method");
24         super.show();
25         System.out.println("value2 from super class =
   "+super.value2);
26         System.out.println("value3 = "+value3);
27         System.out.println("value4 = "+value4);
28     }
29 }
30
```

Terminal Test cases

< Prev Reset Submit Next >

39.1.3. Write a Java program to Access the Class members using super Keyword

12:05

Write a Java program to access the class members using **super** Keyword.

Create a class called `SuperClass` with the below members:

- declare two member fields **value1** and **value2** of type **int**
- a parameterized constructor with **two** arguments, which assigns two arguments to the members respectively
- a method called **show()** which will print **This is super class show() method** as well as the value of **value1**.

Create another class called `SubClass` which is derived from the class `SuperClass`, and has the below members:

- declare two member fields **value3** and **value4** of type **int**
- a parameterized constructor with **four** arguments, which assigns the first two arguments with **SuperClass** members and next two values with **SubClass** members
- a method called **show()** which
  1. will print **This is sub class show() method**
  2. will call **show()** of **SuperClass**
  3. will print **value2** from **SuperClass**
  4. will print **value3** of **SubClass**
  5. will print **value4** of **SubClass**

Write a class `AccessUsingSuper` with the **main()** method, create an object to `SubClass` which calls the method **show()**.

For example, if the input is given as [10, 20, 30, 40] then the output should be:

```
This is sub class show() method
This is super class show() method
value1 = 10
```

Sample Test Cases +

Explorer AccessU... Submit Plots

```
9   void show(){
10      System.out.println("This is super class show() method");
11      System.out.println("value1 = "+value1);
12  }
13  }
14
15  class SubClass extends SuperClass {
16      int value3, value4;
17      SubClass(int value1,int value2,int value3,int value4){
18          super(value1,value2);
19          this.value3=value3;
20          this.value4=value4;
21      }
22      void show(){
23          System.out.println("This is sub class show() method");
24          super.show();
25          System.out.println("value2 from super class =
    "+super.value2);
26          System.out.println("value3 = "+value3);
27          System.out.println("value4 = "+value4);
28      }
29  }
30
31  public class AccessUsingSuper {
32      public static void main(String[] args) {
33          SubClass obj = new SubClass(Integer.parseInt(args[0]),
    Integer.parseInt(args[1]), Integer.parseInt(args[2]),
    Integer.parseInt(args[3]));
34          obj.show();
35      }
36  }
```

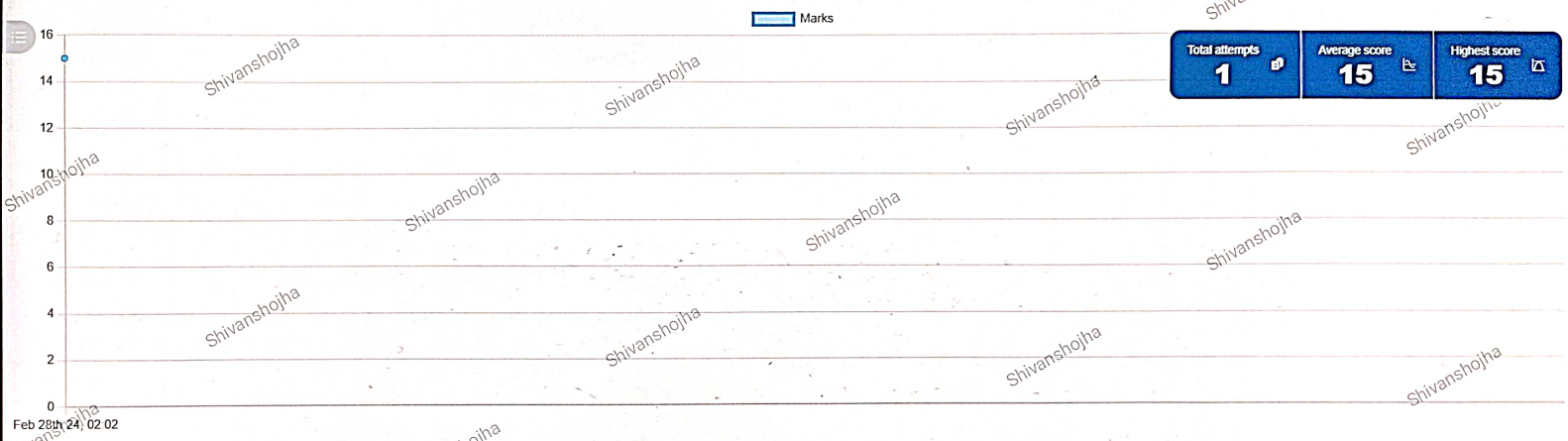
Terminal Test cases

< Prev Reset Submit Next >

Java - Method Overriding - II

Try Again

2 hours 15 marks(s)



Total attempts	Average score	Highest score
1	15	15

Attempt	Test date	Results	Status	Actions
1	February 28th 2024, 2:02:10 am	15/15	Passed	Details

40.1.1. Understanding polymorphism and IS-A relationship

03:26

In Java, since Object class is the superclass (root class) of every class, instance of any class **is also an instance** of Object class.

For example:

```
class Person {
}
Person p = new Person();
```

In the above code, the instance referred by `p` IS-A `Person`. Since every class in Java, including `Person` is a subclass of Object, the statement `p` IS-A Object is also correct.

Which means every object in Java will have more than one IS-A relationship (One with its own class type and one with Object class type).

Any object which satisfies more than one IS-A relation is called polymorphic. For example, let us write classes `A`, `B` and `C` which override **toString()** method of **Object** class.

Observe the given code carefully to understand the polymorphic behaviour and complete the incomplete code.

1. You will notice that `System.out.println(a);` is same as `System.out.println(a.toString());`, because the **println** method which takes any object internally calls `toString()` method on that reference.
2. You will notice that the overridden `toString()` method in class `B` first invokes the `toString()` method present in class `A` using the `super` keyword.
3. Similar is the case with class `C`.
4. Both the overridden methods in `B` and `C` append their own information to the value returned

Sample Test Cases +

Explorer Polymorp... Submit Plots

```
1  package q11276;
2  public class PolymorphismExample2 {
3      public static void main(String[] args) {
4          /* create three objects a, b, c of the
5             classes A, B ,C respectively */
6          A a = new A();
7          B b = new B();
8          C c = new C();
9
10
11         System.out.println(a);
12         System.out.println(b);
13         System.out.println(c.toString());
14     }
15 }
16 class A {
17     public String toString() {
18         return "A";
19     }
20 }
21 class B extends A {
22     public String toString() {
23         return super.toString() + " " + "B";
24     }
25 }
26 class C extends B {
27     public String toString() {
28         return super.toString() + " " + "C";
29     }
30 }
```

Terminal Test cases

Prev Reset Submit Next

40.1.2. Write a Java program that Implements Runtime Polymorphism

03:43

Write a Java program that implements **runtime polymorphism**.

Create a class `Animal` with one method `whoAmI()` which will print **I am a generic animal.**

Create another class `Dog` which extends `Animal`, which will print **I am a dog.**

Create another class `Cow` which extends `Animal`, which will print **I am a cow.**

Create another class `Snake` which extends `Animal`, which will print **I am a snake.**

Write a class `RuntimePolymorphismDemo` with the **main()** method, create objects to all the classes `Animal`, `Dog`, `Cow`, `Snake` and call `whoAmI()` with each object.

Note: Please don't change the package name.

Sample Test Cases +

Explorer RuntimeP... Submit Plots

```
1  package q11277;
2  public class RuntimePolymorphismDemo {
3      public static void main(String[] args) {
4          Animal ref1 = new Animal();
5          Animal ref2 = new Dog();
6          Animal ref3 = new Cow();
7          Animal ref4 = new Snake();
8          ref1.whoAmI();
9          ref2.whoAmI();
10         ref3.whoAmI();
11         ref4.whoAmI();
12     }
13 }
14 class Animal{
15     void whoAmI(){
16         System.out.println("I am a generic animal");
17     }
18 }
19 class Dog extends Animal{
20     void whoAmI(){
21         System.out.println("I am a dog");
22     }
23 }
24 class Cow extends Animal{
25     void whoAmI(){
26         System.out.println("I am a cow");
27     }
28 }
29 class Snake extends Animal{
30     void whoAmI(){
31         System.out.println("I am a snake");
```

Terminal Test cases

< Prev Reset Submit Next >

41.1.1. Understanding Object class

02:05

In Java **Object** is the root class of all classes. Which means that all the methods of Object class described below are also present in each and every class in Java.

1. **clone()** - creates and returns a copy of this object. However, for this method to work the class has to implement Cloneable interface.
2. **equals(Object obj)** - it compares if two references are pointing to the same address. However, you can override this method to provide custom implementation which will verify the contents and not just references.We will learn more about it in the later sections.
3. **finalize()** - this method is called by the garbage collector on an object when garbage collection determines that there are no more references to the object. Programmers do not call it, but can override and write cleanup code in it.
4. **getClass()** - this method returns the in-memory runtime representation of the **Class** object that was loaded and used to create the instance. For example:

```
Student st = new Student();
Class clazz = st.getClass();
```

   The reference **clazz** points to the Class object which has the details of the Student class loaded in memory.
5. **hashCode()** - this method returns a hash code value (an integer) for the object. (we will learn more about it in data structures)
6. **toString()** - this method returns the string representation of the object.

There are many other methods like **notify**, **notifyAll**, and **wait** which are used to synchronize data access and method calls when multiple threads are involved.

Can we directly create an object of the **Object** class using the **new** keyword?

- Yes, it is allowed
- (●) No, it is not allowed

42.1.1. Understanding the toString method in Object class

06:59

The `toString` present in the `Object` class ensures that an object of any class in Java can be converted into a String representation.

If we do not override the `toString` method, by default the `toString` method present in the Object's class will be called (which is of no much use).

Observe the Code given in the Editor

- The **ToStringExample** class demonstrates how the **toString()** method can be used to obtain a customized string representation of an object.
- It concatenates the string **a.toString() :** with the result of the **toString()** method of the A class.

Now, Complete the code for a **class A** that matches the following requirements

1. The class should have a private integer instance variable.
2. The class should have a constructor that takes an integer parameter and initializes the instance variable.
3. The class should override the **toString()** method inherited from the Object class.
4. The **toString()** method should return a string in the format: **"The value is : "** where represents the value of the instance variable.

Note: Please don't change the package name.

Sample Test Cases +

Explorer ToStringE... Submit Plots

```
1   package q11279;
2   public class ToStringExample {
3       public static void main(String[] args) {
4           A a = new A(4);
5           System.out.println("a.toString() : " + a.toString());
6           System.out.println("a : " + a);
7       }
8   }
9   class A{
10      private int var;
11      A(int val){
12          this.var=val;
13      }
14      public String toString(){
15          return "The value is : "+var;
16      }
17      
18  }
```

Terminal Test cases

Prev Reset Submit Next

42.1.2. Understanding the equals method in Object class

20:09

If we do not override the `equals` method, by default the equals method present in the Object's class will be called.

The implementation of `equals` present in the root class Object, only verifies if the references are the same. It does not verify if the contents are the same. Content verification is the responsibility of the overriding implementation.

There is an incomplete code provided to you in the Editor, follow the comment lines in order to complete the code in the Main class and The functionality of class A is described below:

- `class A` is a simple class with a private int field named value and a constructor that takes an `int` parameter to initialize the value field.
- The class also defines an `equals()` method that overrides the default `equals()` method inherited from the Object class.
- If the objects are not the same instance, it checks if the `otherObject` is an instance of class A using the `instanceof` operator
  1. If `otherObject` is an instance of class A, it casts `otherObject` to type A and assigns it to a variable `otherARef`
  2. It then compares the value of the current object (this.value) with the value of `otherARef`.If the values are the same, it means the objects are equal, so it returns true.

Now complete the code in the Editor By following the above statements and Observe the Output.

Note:

- In the main() method, the code prompts the user to input values for a_1, a_2, and a_3, which are used to create three instances of class A. It then calls the equals() method on a1 to compare it with a2

Sample Test Cases +

Explorer EqualsEx... Submit Plots

```
1  package q11280;
2  import java.util.*;
3  public class EqualsExample {
4      public static void main(String[] args) {
5          Scanner sc= new Scanner(System.in);
6          System.out.print("a1: ");
7          int a_1= sc.nextInt();
8          System.out.print("a2: ");
9          int a_2=sc.nextInt();
10         System.out.print("a3: ");
11         int a_3=sc.nextInt();
12         /*create three instances of class A named a1,a2,a3
13         with the user given values and compare a1 with
14         a2 and a3 using equals() method*/
15         A a1 = new A(a_1);
16         A a2 = new A(a_2);
17         A a3 = new A(a_3);
18         System.out.println("a1.equals(a2) : "+a1.equals(a2));
19         System.out.println("a1.equals(a3) : "+a1.equals(a3));
20         System.out.print("b1: ");
21         int b_1= sc.nextInt();
22         System.out.print("b2: ");
23         int b_2=sc.nextInt();
24         System.out.print("b3: ");
25         int b_3=sc.nextInt();
26         /*create three instances of class B named
27         b1,b2,b3 with the user given values and
28         compare b1 with b2 and b3 using equals() method*/
29         B b1 = new B(b_1);
30         B b2 = new B(b_2);
31         B b3 = new B(b_3);
```

Terminal Test cases

< Prev Reset Submit Next >

42.1.2. Understanding the equals method in Object class

20:09

If we do not override the `equals` method, by default the equals method present in the Object's class will be called.

The implementation of `equals` present in the root class Object, only verifies if the references are the same. It does not verify if the contents are the same. Content verification is the responsibility of the overriding implementation.

There is an incomplete code provided to you in the Editor, follow the comment lines in order to complete the code in the Main class and The functionality of class A is described below:

- `class A` is a simple class with a private int field named value and a constructor that takes an `int` parameter to initialize the value field.
- The class also defines an `equals()` method that overrides the default `equals()` method inherited from the Object class.
- If the objects are not the same instance, it checks if the `otherObject` is an instance of class A using the `instanceof` operator
    1. If `otherObject` is an instance of class A, it casts `otherObject` to type A and assigns it to a variable `otherARef`
    2. It then compares the value of the current object (this.value) with the value of `otherARef`.If the values are the same, it means the objects are equal, so it returns true.

Now complete the code in the Editor By following the above statements and Observe the Output.

Note:

- In the main() method, the code prompts the user to input values for a_1, a_2, and a_3, which are used to create three instances of class A. It then calls the equals() method on a1 to compare it with a2

Sample Test Cases +

Explorer EqualsEx... Submit Plots

```
27      b1,b2,b3 with the user given values and
28      compare b1 with b2 and b3 using equals() method*/
29      B b1 = new B(b_1);
30      B b2 = new B(b_2);
31      B b3 = new B(b_3);
32      System.out.println("b1.equals(b2) : "+b1.equals(b2));
33      System.out.println("b1.equals(b3) : "+b1.equals(b3));
34
35  }
36  }
37  class A {
38      private int value;
39      public A(int value) {
40          this.value = value;
41      }
42      // complete the code
43      public boolean equals(Object obj){
44          if(this==obj)
45              return true;
46          if(!(obj instanceof A))
47              return false;
48          A obj1=(A) obj;
49          return this.value==obj1.value;
50      }
51  }
52  class B {
53      private int value;
54      public B(int value) {
55          this.value = value;
56      }
57  }
```

Terminal Test cases

Prev Reset Submit Next

42.2.1. Understanding scope

01:13

Scope can be defined as a portion or block of code in which a variable is visible.

We apply the word scope to variables/references and not so much to methods as methods are members of classes and interfaces.

Variables and references can be declared in:

1. Blocks - these can be if/else/switch, for/while loops, methods and constructors or named blocks. These are also called **local variables**. For example:

```
public static void main(String[] args) {
    int x = Integer.parseInt(args[0]); // x is visible only inside this main method
    if (x > 1) { //if-block-start
        int y = 2 * x;  // y is visible only inside this if block
        System.out.println("y = " + y);
    } //if-block-end
}
```

2. Method and Constructor parameters. For example:

```
public int sum(int num1, int num2) { //method-block-start
    return num1 + num2; // num1 and num2 are only visible inside this method block
} //method-block-end
```

3. Class - as instance fields

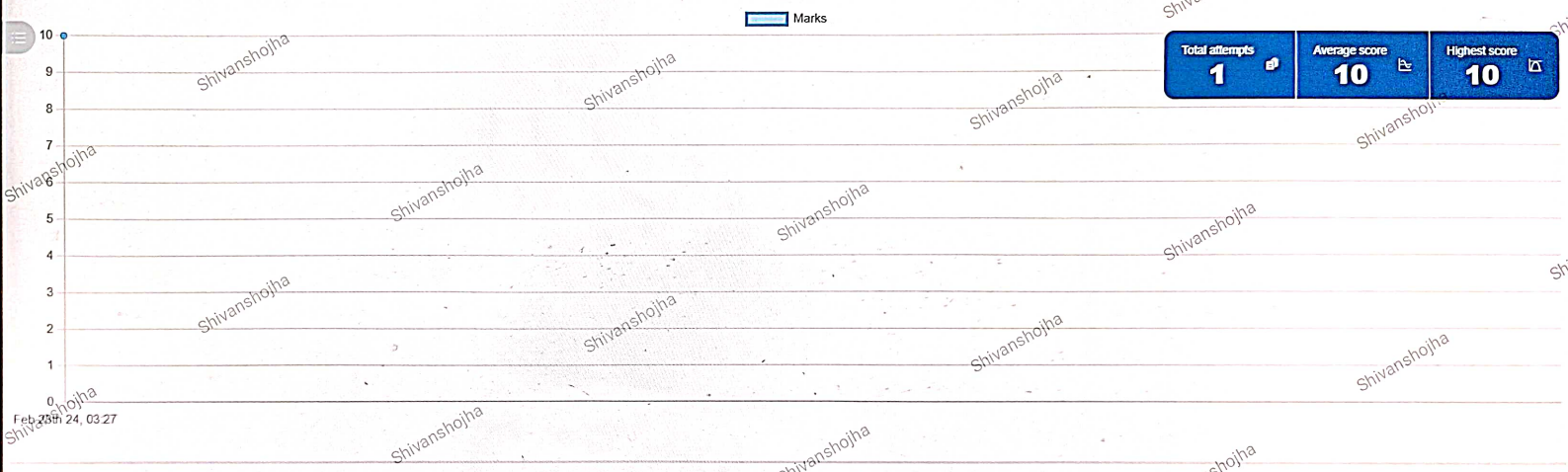
```
class A { //class-block-start
    private String name; //reference name is visible anywhere inside the class block
    public String getName() {
        return name; // example usage 1
    }
    public String toString() {
        return "A [ name = " + name + " ]"; // example usage 2
    }
} //class-block-end
```

- Statement 5 is in the wrong location, it should have been declared above statement 1.
- Statement 2 will give a compilation error, because there is no variable called `this` declared.
- ☑ Statements 3 and 4 will produce compiler errors saying `value2` cannot be resolved, since `value2` is neither declared locally in the enclosing block nor as a field in the class.
- Statement 5 will produce a compilation error because `value1` is not initialized.

Java - Unit 3 - Industry Specific Questions

Try Again

1 hour 10 marks(s)



Total attempts	Average score	Highest score
1	10	10

Attempt	Test date	Results	Status	Actions
1	February 28th 2024, 3:27:26 am	10/10	Passed	Details

43.1.1. Understanding Interfaces

01:25

As the meaning of the word `interface` suggests, it is the point where two systems or subjects meet to interact.

Interface holds the same meaning in object oriented programming languages too. An `interface` defines a contract using which two classes/programs/systems can interact with each other.

In Java, `interface` is like a `class`, it is a reference type. It can have constants (they are not called fields), method signatures and nested members (we will learn more about nested members later).

The methods declared in interfaces should not contain the method body.
[Note: In **Java 8** and later versions, interfaces can contain default and static methods which can contain method body. We will learn more about them in later sections.]
Only the method signature should be present with a semicolon as terminator. For example:

```
interface Person {
    public static final int RETIREMENT_AGE = 60;  // example of a constant
    void setName(String name);  // only method signature without method body
    String getName();
    void setAge(int age);
    int getAge();
}
```

All methods declared in an interface are by default public and hence, we did not use the keyword `public` in the method signatures.

Interfaces cannot be instantiated. Meaning, Java compiler will throw out an error for a statement like : `Person p = new Person();`.

We can instantiate classes that implement the interface. Meaning, we can instantiate classes that provide the implementation for all the methods declared in the interface. For example:

```
public class Teacher implements Person {
    private String name;
```

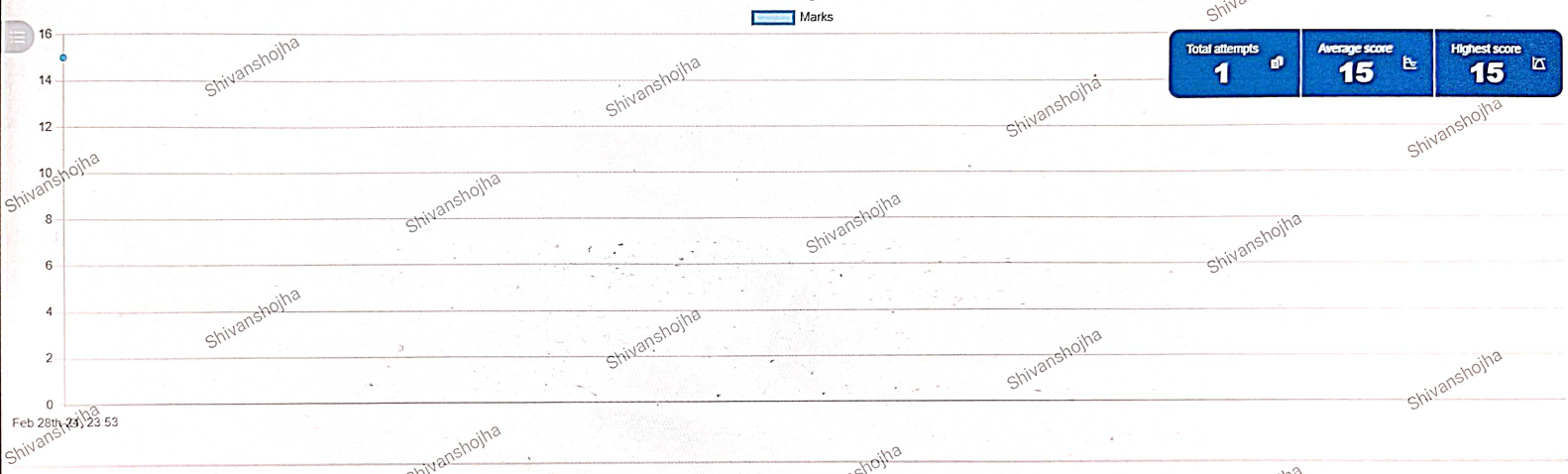
- [ ] Statement 1 will result in compilation error, since class `Teacher` cannot implement two interfaces at the same time.
- [x] Statement 2 **will result** in a compilation error because we cannot instantiate an interface.
- [ ] Statement 3 **will not result** in a compilation error because we are trying to instantiate `Person` and assign to `Citizen`.
- [x] Statements 4 and 5 do not result in compilation errors.

Home Learn Anywhere 12216507.st@lpu.in Support Logout

Java - Interfaces - I

Try Again

2 hours 15 marks(s)



Total attempts	Average score	Highest score
1	15	15

Attempt	Test date	Results	Status	Actions
1	February 28th 2024, 11:53:53 pm	15/15	Passed	Details

< Prev Next >

44.1.1. Understanding the usage of Interfaces

06:33

An interface is primarily used to create a type.

In Java, a class is also a type which has behaviour attached to it.

However, in large object-oriented systems where multiple classes interact with each other an interface is the preferred means of creating a type. Classes are still needed which implement the type, however the publicly visible contract (methods) is published by the interface and the implementation is provided in the implementing classes.

By doing this we achieve what is called **loose-coupling**, where a class does not depend on the actual implementation (of another class) but rather depends on the contract (i.e. the methods) published through the interface. This ensures that the classes which rely on the interface, need not change whenever the underlying implementation in the implementing classes changes.

See and understand the below code to know how the class InterfaceDemo need not know the difference in the implementation present in classes A and B, when it uses the interface.

```
public class InterfaceDemo {
    public static void main(String[] args) {
        Greeting g1 = new A();
        Greeting g2 = new B();
        System.out.println(g1.getGreetings("Thor"));
        System.out.println(g2.getGreetings("Thor"));
    }
}
interface Greeting {
    String getGreetings(String name);
}
```

Sample Test Cases +

Explorer

Interface...

Submit

Plots

```
1  import java.util.*;
2
3  interface Printable {
4      void print();
5  }
6
7  // Implement the interface in a class
8  class Document implements Printable {
9      public void print() {
10         Scanner sc = new Scanner(System.in);
11         System.out.print("Enter the first value: ");
12         int num=sc.nextInt();
13         System.out.print("Enter the second value: ");
14         int num1=sc.nextInt();
15         System.out.println("You entered: "+num+" and "+num1);
16     }
17 }
18
19
20 public class InterfaceDemo {
21     public static void main(String[] args) {
22         // Create an instance of the class implementing the
   interface
23         Printable document = new Document();
24
25         // Call the method defined in the interface
26         document.print();
27     }
28 }
```

Terminal Test cases

< Prev Reset Submit Next >

44.1.2. Write a Java program to implement Interface

19:36

Write a Java program that implements an **interface**.

Create an interface called `Car` with two abstract methods `String getName()` and `int getMaxSpeed()`. Also declare one **default** method `void applyBreak()` which has the code snippet

```
System.out.println("Applying break on " + getName());
```

In the same interface include a **static** method `Car getFastestCar(Car car1, Car car2)`, which returns **car1** if the **maxSpeed** of **car1** is greater than or equal to that of **car2**, else should return **car2**.

Create a class called `BMW` which implements the interface `Car` and provides the implementation for the abstract methods **getName()** and **getMaxSpeed()** (make sure to declare the appropriate fields to store **name** and **maxSpeed** and also the constructor to initialize them).

Similarly, create a class called `Audi` which implements the interface `Car` and provides the implementation for the abstract methods **getName()** and **getMaxSpeed()** (make sure to declare the appropriate fields to store **name** and **maxSpeed** and also the constructor to initialize them).

Create a **public** class called `MainApp` with the **main()** method.
Take the input from the command line arguments. Create objects for the classes `BMW` and `Audi` then print the fastest car.

Note:
Java 8 introduced a new feature called `default` methods or `defender` methods, which allow developers to add new methods to the interfaces without breaking the existing implementation of these

Sample Test Cases +

Explorer

MainApp....

Submit

Plots

```
1  package q11284;
2  interface Car {
3      String getName();
4      int getMaxSpeed();
5      default void applyBreak(){
6          System.out.println("Applying break on "+getName());
7      }
8      static void getFastestCar(Car car1,Car car2){
9          System.out.print("Fastest car is : ");
10         if(car1.getMaxSpeed()>=car2.getMaxSpeed())
11             System.out.println(car1.getName());
12         else
13             System.out.println(car2.getName());
14     }
15 }
16 class BMW implements Car {
17     String name;
18     int speed;
19     public String getName(){
20         return name;
21     }
22     public int getMaxSpeed(){
23         return speed;
24     }
25     BMW(String name,int speed){
26         this.name=name;
27         this.speed=speed;
28     }
29 }
30 class Audi implements Car {
31     String name;
```

Terminal Test cases

< Prev Reset Submit Next >

44.1.2. Write a Java program to Implement Interface

19:36

Write a Java program that implements an **interface**.

Create an interface called `Car` with two abstract methods `String getName()` and `int getMaxSpeed()`. Also declare one **default** method `void applyBreak()` which has the code snippet

```
System.out.println("Applying break on " + getName());
```

In the same interface include a **static** method `Car getFastestCar(Car car1, Car car2)`, which returns **car1** if the **maxSpeed** of **car1** is greater than or equal to that of **car2**, else should return **car2**.

Create a class called `BMW` which implements the interface `Car` and provides the implementation for the abstract methods **getName()** and **getMaxSpeed()** (make sure to declare the appropriate fields to store **name** and **maxSpeed** and also the constructor to initialize them).

Similarly, create a class called `Audi` which implements the interface `Car` and provides the implementation for the abstract methods **getName()** and **getMaxSpeed()** (make sure to declare the appropriate fields to store **name** and **maxSpeed** and also the constructor to initialize them).

Create a **public** class called `MainApp` with the **main()** method.
Take the input from the command line arguments. Create objects for the classes `BMW` and `Audi` then print the fastest car.

Note:
Java 8 introduced a new feature called `default` methods or `defender` methods, which allow developers to add new methods to the interfaces without breaking the existing implementation of these

Sample Test Cases +

Explorer

MainApp....

Submit

Plots

```
25      BMW(String name,int speed){
26          this.name=name;
27          this.speed=speed;
28      }
29  }
30  class Audi implements Car {
31      String name;
32      int speed;
33      public String getName(){
34          return name;
35      }
36      public int getMaxSpeed(){
37          return speed;
38      }
39      Audi(String name,int speed){
40          this.name=name;
41          this.speed=speed;
42      }
43  }
44  public class MainApp {
45      public static void main(String args[]) {
46          String name1=args[0];
47          int speed1=Integer.parseInt(args[1]);
48          String name2=args[2];
49          int speed2=Integer.parseInt(args[3]);
50          Car c1 = new BMW(name1,speed1);
51          Car c2 = new Audi(name2,speed2);
52          Car.getFastestCar(c1,c2);
53      }
54  }
55
```

Terminal Test cases

< Prev Reset Submit Next >

Home Learn Anywhere 12216507.st@lpu.in Support Logout

Java - Interfaces - II Try Again

2 hours 15 marks(s)

Total attempts	Average score	Highest score
1	15	15

Attempt	Test date	Results	Status	Actions
1	February 29th 2024, 1:08:08 am	15/15	Passed	Details

45.1.1. Write a Java program to find Areas of different Shapes using abstract class

23:47

Write a Java program to illustrate the **abstract class** concept.

Create an abstract class `CalcArea` and declare the methods **triangleArea(double b, double h)**, **rectangleArea(double l, double b)**, **squareArea(double s)**, **circleArea(double r)**.

Create a class `FindArea` which extends the abstract class `CalcArea` used to find areas of triangle, rectangle, square, circle.

Write a class `Area` with the **main()** method which will receive **two** arguments and convert them to **double** type.

If the input is given as command line arguments to the **main()** as **"1.2","2.7"** then the program should print the output as:

```
Area of triangle : 1.62
Area of rectangle : 3.24
Area of square : 1.44
Area of circle : 22.890600000000006
```

Note: Please don't change the package name.

Sample Test Cases +

Explorer

Area.java

Submit

```
1  package q11286;
2  public class Area {
3      public static void main(String args[]) {
4          FindArea area = new FindArea();
5          area.triangleArea(Double.parseDouble(args[0]),
   Double.parseDouble(args[1]));
6          area.rectangleArea(Double.parseDouble(args[0]),
   Double.parseDouble(args[1]));
7          area.squareArea(Double.parseDouble(args[0]));
8          area.circleArea(Double.parseDouble(args[1]));
9      }
10 }
11 // Write all the classes with definitions
12 abstract class CalcArea{
13     abstract void triangleArea(double b, double h);
14     abstract void rectangleArea(double l, double b);
15     abstract void squareArea(double s);
16     abstract void circleArea(double r);
17 }
18 class FindArea extends CalcArea{
19 void triangleArea(double b,double h){
20     double area=0.5*b*h;
21     System.out.println("Area of triangle : "+area);
22     }
23 void rectangleArea(double l,double b){
24     double area=l*b;
25     System.out.println("Area of rectangle : "+area);
26     }
27 void squareArea(double s){
28     double area=s*s;
29     System.out.println("Area of square : "+area);
```

Terminal Test cases

< Prev Reset Submit Next >

45.1.1. Write a Java program to find Areas of different Shapes using abstract class

23:47

Write a Java program to illustrate the **abstract class** concept.

Create an abstract class `CalcArea` and declare the methods **triangleArea(double b, double h)**, **rectangleArea(double l, double b)**, **squareArea(double s)**, **circleArea(double r)**.

Create a class `FindArea` which extends the abstract class `CalcArea` used to find areas of triangle, rectangle, square, circle.

Write a class `Area` with the **main()** method which will receive **two** arguments and convert them to **double** type.

If the input is given as command line arguments to the **main()** as **"1.2","2.7"** then the program should print the output as:

```
Area of triangle : 1.62
Area of rectangle : 3.24
Area of square : 1.44
Area of circle : 22.890600000000006
```

Note: Please don't change the package name.

Sample Test Cases +

Explorer

Area.java

Submit

```
6       area.rectangleArea(Double.parseDouble(args[0]),
        Double.parseDouble(args[1]));
7       area.squareArea(Double.parseDouble(args[0]));
8       area.circleArea(Double.parseDouble(args[1]));
9   }
10  }
11  // Write all the classes with definitions
12  abstract class CalcArea{
13      abstract void triangleArea(double b, double h);
14      abstract void rectangleArea(double l, double b);
15      abstract void squareArea(double s);
16      abstract void circleArea(double r);
17  }
18  class FindArea extends CalcArea{
19  void triangleArea(double b,double h){
20      double area=0.5*b*h;
21      System.out.println("Area of triangle : "+area);
22      }
23  void rectangleArea(double l,double b){
24      double area=l*b;
25      System.out.println("Area of rectangle : "+area);
26      }
27  void squareArea(double s){
28      double area=s*s;
29      System.out.println("Area of square : "+area);
30      }
31  void circleArea(double r){
32      double area=3.14*r*r;
33      System.out.println("Area of circle : "+area);
34      }
35  }
```

Terminal Test cases

< Prev Reset Submit Next >

45.1.2. Write a Java program to illustrate the abstract class concept

08:55

Write a Java program to illustrate the **abstract class** concept.

Create an abstract class `Shape`, which contains an empty method **numberOfSides()**.

Define three classes named `Trapezoid`, `Triangle` and `Hexagon` extends the class `Shape`, such that each one of the classes contains only the method **numberOfSides()**, that contains the number of sides in the given geometrical figure.

Write a class `AbstractExample` with the **main()** method, declare an object to the class `Shape`, create instances of each class and call numberOfSides() methods of each class.

Sample Input and Output:

```
Number of sides in a trapezoid are 4
Number of sides in a triangle are 3
Number of sides in a hexagon are 6
```

Note: Please don't change the package name.

Sample Test Cases +

Explorer

Abstract...

Submit

Plots

```
1  package q11287;
2  abstract class Shape{
3      abstract void numberOfSides();
4  }
5  class Trapezoid extends Shape{
6      void numberOfSides(){
7          System.out.println("Number of sides in a trapezoid are
   4");
8      }
9  }
10 class Triangle extends Shape{
11     void numberOfSides(){
12         System.out.println("Number of sides in a triangle are 3");
13     }
14 }
15 class Hexagon extends Shape{
16     void numberOfSides(){
17         System.out.println("Number of sides in a hexagon are 6");
18     }
19 }
20
21 public class AbstractExample {
22     public static void main(String[] args) {
23         Shape s;
24         s = new Trapezoid();
25         s.numberOfSides();
26         s = new Triangle();
27         s.numberOfSides();
28         s = new Hexagon();
29         s.numberOfSides();
30     }
```

Terminal Test cases

45.1.3. What is an abstract class?

17:12

In Java, a class when declared with abstract keyword becomes an abstract class. For example:

```
public abstract class A {
}
```

As per the dictionary, **abstract** means **not concrete**. In object oriented languages also, when a class is marked with abstract keyword, it indicates that the class is not concrete. **It cannot be instantiated.** Meaning, the Java compiler will give an error, saying abstract class `A` cannot be instantiated, if we write a statement like `A a = new A();`.

Abstract classes like normal (concrete) classes can have fields, constructors and methods.

Abstract classes can also have one or more abstract methods. An abstract method should have the keyword `abstract` in its signature and it should not have a method body. For example:

```
public abstract int sum(int num1, num2);
```

A concrete class is not allowed to have a method declared as `abstract`. Java compiler will throw an compilation error if an attempt is made to declare an abstract method in a concrete class.

In interfaces, the method declarations with signatures and without method bodies are by default marked `abstract`. However, in a abstract class we have to mark a method explicitly with `abstract` keyword.

Abstract classes are present so only to be inherited by other classes. Abstract classes usually contain common code that can be shared by the subclasses.

Observe the given code and declare an abstract class **AbstractGreeting** that implements interface

Sample Test Cases +

Explorer Abstract... Submit Plots

```
1  package q35957;
2  public class AbstractDemo {
3      public static void main(String[] args) {
4          Greeting english = new EnglishGreeting();
5          Greeting spanish = new SpanishGreeting();
6          System.out.println(english.getStandardMessage("Winston"));
7          System.out.println(english.getCustomMessage("Winston"));
8          System.out.println(spanish.getStandardMessage("Martin"));
9          System.out.println(spanish.getCustomMessage("Martin"));
10     }
11 }
12 interface Greeting {
13     public String getStandardMessage(String name);
14     public String getCustomMessage(String name);
15 }
16
17 //Define the required abstract class here
18 abstract class AbstractGreeting implements Greeting{
19     // abstract String getCustomMessage(String name);
20 }
21 class EnglishGreeting extends AbstractGreeting {
22     public String getCustomMessage(String name) {
23         return "Hello " + name;
24     }
25     public String getStandardMessage(String name){
26         return "Hi "+name;
27     }
28 }
29 class SpanishGreeting extends AbstractGreeting {
30     public String getCustomMessage(String name) {
31         return "Holla " + name;
```

Terminal Test cases

45.1.3. What is an abstract class?

17:12

In Java, a class when declared with abstract keyword becomes an abstract class. For example:

```
public abstract class A {
}
```

As per the dictionary, **abstract** means **not concrete**. In object oriented languages also, when a class is marked with abstract keyword, it indicates that the class is not concrete. **It cannot be instantiated**. Meaning, the Java compiler will give an error, saying abstract class `A` cannot be instantiated, if we write a statement like `A a = new A();`.

Abstract classes like normal (concrete) classes can have fields, constructors and methods.

Abstract classes can also have one or more abstract methods. An abstract method should have the keyword `abstract` in its signature and it should not have a method body. For example:

```
public abstract int sum(int num1, num2);
```

A concrete class is not allowed to have a method declared as `abstract`. Java compiler will throw an compilation error if an attempt is made to declare an abstract method in a concrete class.

In interfaces, the method declarations with signatures and without method bodies are by default marked `abstract`. However, in a abstract class we have to mark a method explicitly with `abstract` keyword.

Abstract classes are present so only to be inherited by other classes. Abstract classes usually contain common code that can be shared by the subclasses.

Observe the given code and declare an abstract class **AbstractGreeting** that implements interface

Sample Test Cases +

Explorer

Abstract...

Submit

Plots

```
6       System.out.println(english.getStandardMessage("Winston"));
7       System.out.println(english.getCustomMessage("Winston"));
8       System.out.println(spanish.getStandardMessage("Martin"));
9       System.out.println(spanish.getCustomMessage("Martin"));
10      }
11  }
12  interface Greeting {
13      public String getStandardMessage(String name);
14      public String getCustomMessage(String name);
15  }
16
17  //Define the required abstract class here
18  abstract class AbstractGreeting implements Greeting{
19      // abstract String getCustomMessage(String name);
20  }
21  class EnglishGreeting extends AbstractGreeting {
22      public String getCustomMessage(String name) {
23          return "Hello " + name;
24      }
25      public String getStandardMessage(String name){
26          return "Hi "+name;
27      }
28  }
29  class SpanishGreeting extends AbstractGreeting {
30      public String getCustomMessage(String name) {
31          return "Holla " + name;
32      }
33      public String getStandardMessage(String name){
34          return "Hi "+name;
35      }
36  }
```

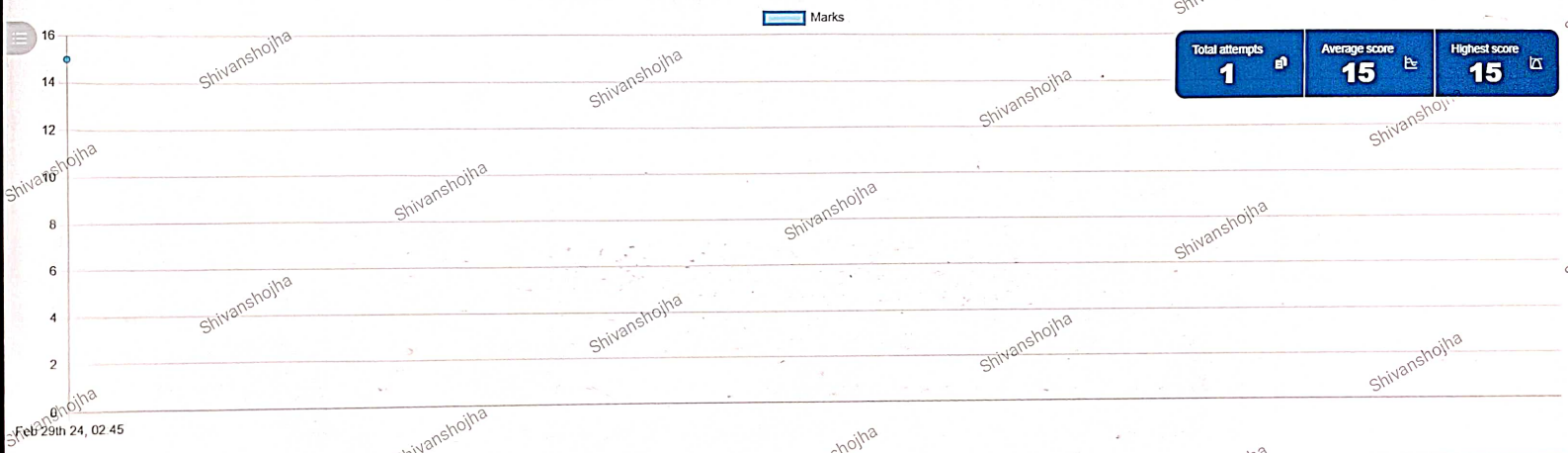
Terminal Test cases

Home Learn Anywhere 12216507.st@lpu.in Support Logout

Java - Abstract Class

Try Again

2 hours 15 marks(s)



Total attempts	Average score	Highest score
1	15	15

Attempt	Test date	Results	Status	Actions
1	February 29th 2024, 2:45:18 am	15/15	Passed	Details

46.1.1. Default methods in Interfaces

06:58

Default methods in interfaces:

Before Java 8, interfaces could have only abstract methods. The implementation of these methods has to be provided in a separate class. So, if a new method is to be added to an interface, its implementation code must be provided in the class implementing the same interface. To overcome this issue, Java 8 has introduced the concept of default methods, allowing the interfaces to have methods with implementation without affecting the classes that implement the interface.

Let's go through a clear example to illustrate the use of default methods in Java interfaces. Suppose you have an interface called `Shape` representing different geometric shapes, and you want to add a method to calculate the area. However, adding this method to the interface might break existing implementations. Default methods come to the rescue in this scenario.

```
public interface Shape {
  // Abstract method (to be implemented by concrete classes)
  void draw();

  // Default method to calculate area
  default double calculateArea() {
    return 0.0; // Default implementation for simplicity
  }
}
```

Here, `Shape` has an abstract method `draw()` that concrete classes must implement. Additionally, there's a default method `calculateArea()` with a default implementation returning 0.0. Concrete classes can choose to override this method.

Now, let's create a class `Circle` that implements the `Shape` interface:

Sample Test Cases +

Explorer

Animal.java Dog.java Main.java Submit

```
1  public interface Animal {
2  ...
3  void makeSound();
4  default String eatFood(){
5  ⟶return "eating generic food";
6  }
7  }
8
```

Terminal Test cases

< Prev Reset Submit Next >

46.1.1. Default methods in Interfaces

06:58

Default methods in interfaces:

Before Java 8, interfaces could have only abstract methods. The implementation of these methods has to be provided in a separate class. So, if a new method is to be added to an interface, its implementation code must be provided in the class implementing the same interface. To overcome this issue, Java 8 has introduced the concept of default methods, allowing the interfaces to have methods with implementation without affecting the classes that implement the interface.

Let's go through a clear example to illustrate the use of default methods in Java interfaces.
Suppose you have an interface called `Shape` representing different geometric shapes, and you want to add a method to calculate the area. However, adding this method to the interface might break existing implementations. Default methods come to the rescue in this scenario.

```
public interface Shape {
  // Abstract method (to be implemented by concrete classes)
  void draw();

  // Default method to calculate area
  default double calculateArea() {
    return 0.0; // Default implementation for simplicity
  }
}
```

Here, `Shape` has an abstract method `draw()` that concrete classes must implement. Additionally, there's a default method `calculateArea()` with a default implementation returning 0.0. Concrete classes can choose to override this method.
Now, let's create a class `Circle` that implements the `Shape` interface:

Sample Test Cases +

Explorer

Animal.java Dog.java Main.java Submit

```
1  public class Dog implements Animal{
2      public void makeSound(){
3          System.out.println("Dog barks: Woof! Woof!");
4      }
5      public String eatFood(){
6          return "Dog is eating bones";
7      }
8  }
9
10
```

Terminal Test cases

< Prev Reset Submit Next >