Example -6 : Inheritance and overriding methods

File : **Bicycle.java**

```
//  Bicycle base class
public class Bicycle
{
   // Bicycle class has four fields
   private int cadence;    private int gear;     private int speed;
   private int id;
   private static int numberOfBicycles = 0;

   // Bicycle class has one constructor
   public Bicycle(int startCadence, int startSpeed, int startGear)
   {
      gear = startGear;   cadence = startCadence;      speed = startSpeed;
      id = ++numberOfBicycles;      //  id is an auto incremented field
   }

   //  The fields are private, so to access those fields from other classes , we define getter and setter
   public static int getNumberOfBicycles()      { return numberOfBicycles;  }
   public int getID()                            {  return id;   }
   public int getCadence()                       { return cadence;    }
   public void setCadence(int newValue)          {  cadence = newValue;  }
   public int getGear()                          {  return gear; }
   public void setGear(int newValue)             { gear = newValue;  }
   public int getSpeed()                         { return speed;  }


   //  Other methods of Bicycle
   public void applyBrake(int decrement)      { speed -= decrement;   }
   public void speedUp(int increment)         {   speed += increment;    }

   //  Method for displaying all the data currently stored in an instance.
   public void printDescription()
   {
      System.out.println("\nBike is " + "in gear " + this.gear   +   " with a cadence of " + this.cadence +
                " and travelling at a speed of " + this.speed + ". ");
   }
}
```

File : **RoadBike.java**

```
public class RoadBike extends Bicycle {

   private int tireWidth;    // In millimeters (mm)
```

```
   public RoadBike(int startCadence,  int startSpeed,  int startGear, int newTireWidth)  {
      super(startCadence,  startSpeed,  startGear);
      this.setTireWidth(newTireWidth);
   }
   public int getTireWidth()   {      return this.tireWidth;    }
   public void setTireWidth(int newTireWidth)
   {
      this.tireWidth = newTireWidth;
   }
   @Override
   public void printDescription()   {
      super.printDescription();
      System.out.println("The RoadBike" + " has " + getTireWidth() + " MM tires.");
   }
}
```

File : **MountainBike.java**

```
public class MountainBike extends Bicycle {
   private String suspension;      // the MountainBike subclass adds one field

   // The MountainBike subclass has one constructor
   public MountainBike(  int startCadence, int startSpeed, int startGear,  String suspensionType )
   {
      //  The constructor of the superclass is invoked from the subclass.
      super(startCadence,   startSpeed, startGear);
      this.setSuspension(suspensionType);
   }

   public String getSuspension()                 {    return this.suspension;     }
   public void setSuspension(String suspensionType) {   this.suspension = suspensionType;     }

   //  The overridden printDescription method will call the super method and in addition data about
   //   the suspension is included to the output.
   @Override
   public void printDescription()   {
      super.printDescription();
      System.out.println("The " + "MountainBike has a" + getSuspension() + " suspension.");
   }

}
```

File : **JavaApp1.java**

```
private static void driverInheritanceBicycle()
  {
     Bicycle bike01, bike02, bike03;

     //  Variables referencing the superclass can also refer the subclass
     bike01 = new Bicycle(20, 10, 1);
     bike02 = new MountainBike(20, 10, 5, "Dual");
     bike03 = new RoadBike(40, 20, 8, 23);

     //  JVM calls the appropriate method for the object that is referred to
     //  in each variable. It does not call the method that is defined by the variable's type.
     //  This behavior is referred to as virtual method invocation an important polymorphism
         feature.

     bike01.printDescription();
     bike02.printDescription();
     bike03.printDescription();
  }
```

Example -7 : Inheritance of Interface and creating class using interface and base class

<table>
<tr>
<td>

File : **ISports.java**

```
public interface ISports {
   public void setHomeTeam(String name);
   public void setVisitingTeam(String name);
}
```

File : **IHockey.java**

```
public interface IHockey extends ISports {

   public String playName ="Hockey";
   public void homeGoalScored(int sc);
   public void visitingGoalScored(int sc );
   public void endOfPeriod(int period);
   public void overtimePeriod(int ot);
}
```

</td>
<td>

File : **IFootball.java**

```
public interface IFootball extends ISports {

//  playName field is redefined in this interface
    although the same field is also defined in
    Hockey  interface. Although All interface fields
    are finals you can redefine your owns field with
    same in  name while you extending or
     implementing an interface.

   public String playName ="Football";
   public void homeTeamScored(int points);
   public void visitingTeamScored(int points);
   public void endOfQuarter(int quarter);
}
```

</td>
</tr>
<tr>
<td>

File : **Team.java**

```
//  Base class to create derived class FootballTeam
    and HockeyTeam
public class Team {
   private String tmName;
   public void setTmName ( String nm)     {
         this.tmName = nm;    }
```

</td>
<td></td>
</tr>
</table>

public String getTmName ( ) {
 return this.tmName; }
 }	

File : **FootballTeam.java**

```
// It needs to implement all the methods of Football as well as Sports interface
public class FootballTeam extends Team implements IFootball {
    public void setHomeTeam(String name) {
        System.out.println(playName + " Home Team : " + name);
    }
    public void setVisitingTeam(String name) {
        System.out.println(playName + " Visiting Team : " + name);
    }
    public void homeTeamScored(int points) {
        System.out.println(playName + " Home team Score : " + points);
    }
    public void visitingTeamScored(int points) {
        System.out.println(playName + " Visiting team Score : " + points);
    }
    public void endOfQuarter(int quarter) { System.out.println("Quarter " + quarter + " ended"); }

}
```

File : **HockeyTeam.java**

```
public class HockeyTeam extends Team implements IHockey {
  public void setHomeTeam(String name) {
      System.out.println(playName + " Home Team : " + name);
  }
  public void setVisitingTeam(String name) {
      System.out.println(playName + " Visiting Team : " + name);
  }
  public void homeGoalScored(int sc) {
      System.out.println(playName + " Home Team Score : " + sc);
  }
  public void visitingGoalScored(int sc ) {
      System.out.println(playName + " Visiting Team Score : " + sc);
  }
  public void endOfPeriod(int period) {
      System.out.println("End of " + period + " Period");
  }

  public void overtimePeriod(int ot) {
    System.out.println("End of Overtime Period : " + " Period");
  }
}
```

File : **JavaApp1.java**

```
    //  If variable is declared to be the type of an interface, its value can reference
    //  any object that is instantiated from any class that implements the interface.
    //  ftSports of type interface Sports can be used to reference HockeyTeam too
  private static void driverInterfaceEx()
  {
    FootballTeam ftA = new FootballTeam();
    Team ftTeam = new FootballTeam();          //  referenced by a variable of type superclass
    ISports ftSports = new FootballTeam();     //  referenced by a variable of type interface
    IFootball ftFootball = new FootballTeam(); //  referenced by a variable of type interface

    ftA.setTmName("East Bengal");
    ftA.setHomeTeam(ftA.getTmName());
    ftA.setVisitingTeam("Mohan Bagan");
    ftA.homeTeamScored(3);    ftA.visitingTeamScored(4);
    ftA.endOfQuarter(3);     System.out.print("===================" + "\n");

    // Able to call methods in super class Team
    ftTeam.setTmName("Tampa Bay Lightning");
    System.out.println(ftTeam.getTmName());
    System.out.print("===================" + "\n");

    // Able to call all the method defined in the interface Sports
    ftSports.setHomeTeam("New Jersey Devils");
    ftSports.setVisitingTeam("Mohan Bagan");
    System.out.print("===================" + "\n");

    // Able to call all the method defined in the interface Football
    // like setHomeTeam, setVisitingTeam, homeTeamScored, visitingTeamScored, endOfQuarter

    ftFootball.setHomeTeam("Carolina Hurricanes");
    ftFootball.homeTeamScored(10);
    System.out.print("===================" + "\n");

    HockeyTeam hcA = new HockeyTeam();
    hcA.setTmName("Dallas Stars");
    hcA.setHomeTeam(hcA.getTmName());
    hcA.setVisitingTeam("Florida Panthers");
    hcA.homeGoalScored(5);
    hcA.visitingGoalScored(7);
    hcA.endOfPeriod(2);
    hcA.overtimePeriod(5);
    System.out.print("===================" + "\n");
```

```
    // HockeyTeam class is also implemented the Sports interface we can also
    // reference it using ftSports.
    ftSports = new HockeyTeam();
  }
```

Example -8 : Create binary tree of integer numbers and perform various traversals.

File : **BinaryTree.java**

```
public class Node {
   int value;
   Node left;
   Node right;

   Node(int value) {
      this.value = value;
      right = null;
      left = null;
   }
}
```

```
import java.util.LinkedList;
import java.util.Queue;

public class BinaryTree {
  //  Let's add the starting node of our tree, usually called root:

  Node root;

  /**  Inserting Elements
   *   To insert we have to find the place where we want to add a new node in order to keep the tree
        sorted.  We'll follow these rules starting from the root node:
   *   • if the new node's value is lower than the current node's, we go to the left child
   *   • if the new node's value is greater than the current node's, we go to the right child
   *   • when the current node is null, we've reached a leaf node and we can insert the new node in
       that position

   *  First, we'll create a recursive method to do the insertion:
   */
  private Node addRecursive(Node current, int value) {
    if (current == null) {
      return new Node(value);
    }
    if (value < current.value) {    current.left = addRecursive(current.left, value);
    } else if (value > current.value) {
      current.right = addRecursive(current.right, value);
    } else {
      // value already exists
      return current;
    }
    return current;
  }
  //  Next, we'll create the public method that starts the recursion from the root node:
  public void add(int value) {
    root = addRecursive(root, value);
  }
```

```
//  Finding an Element
//  Let's now add a method to check if the tree contains a specific value. As before, we'll first create a
//  recursive method that traverses the tree:
//  Here, we're searching for the value by comparing it to the value in the current node, then
    continue in the left or right child depending on that
private Node containsNodeRecursive(Node current, int value) {
   if (current == null)      {    return null;    }
   if (value == current.value) {    return current;     } //  If value match then return that node
   return value < current.value
      ? containsNodeRecursive(current.left, value) : containsNodeRecursive(current.right, value);
}
//  Create the public method that starts from the root:
public Node containsNode(int value) {
   return containsNodeRecursive(root, value);
}
//  Deleting an Element from the tree.
//  We have to find the node to delete in a similar way as we did before:
private Node deleteRecursive(Node current, int value) {
   if (current == null) {   return null;     }
   if (value == current.value) {
      // Node to delete found
      deleteNode(current);   // delete the node
   }
   if (value < current.value) {
      current.left = deleteRecursive(current.left, value);
      return current;
   }
   current.right = deleteRecursive(current.right, value);
   return current;
}
//   Once we find the node to delete, there are 3 main different cases:
//   • a node has no children – this is the simplest case; we just need to replace this node with null in
       its parent node
//   • a node has exactly one child – in the parent node, we replace this node with its only child.
//   • a node has two children – this is the most complex case because it requires a tree
       reorganization

private Node deleteNode(Node current)
{
   //   Let's see how we can implement the first case when the node is a leaf node:
   if (current.left == null && current.right == null) {   return null;  }

   //   Let's continue with the case when the node has one child.
```

```
    //   We're returning the non-null child so it can be assigned to the parent node.
    if (current.right == null) {     return current.left;    }
    if (current.left == null)  {     return current.right;   }

    //  Finally, we have to handle the case where the node has two children.
    //  We need to find the node that will replace the deleted node.
    //  Then, we assign the smallest value to the node to delete and after that, we'll delete it from the
        right subtree:
    int smallestValue = findSmallestValue(current.right);
    current.value = smallestValue;
    current.right = deleteRecursive(current.right, smallestValue);
    return current;
}
//  Find the node that will replace the deleted node. We'll use the smallest node of the node to be
    deleted's right sub-tree:
private int findSmallestValue(Node root) {
    return root.left == null ? root.value : findSmallestValue(root.left);
}
//  Finally, let's create the public method that starts the deletion from the root:
public void delete(int value) {
    deleteRecursive(root, value);
}
```

// Traversing the Tree - We can traverse the tree in different ways of traversing a tree.

// 1. **Depth-First Search**

// In this type of traversal that goes deep as much as possible in every child before exploring the next sibling. There are several ways to perform a depth-first search: in-order, pre-order and post-order. The in-order traversal consists of first visiting the **left sub-tree, then the root node, and finally the right sub-tree**:

```
public void traverseInOrder(Node node) {
    if (node != null) {
        traverseInOrder(node.left);
        System.out.print(" " + node.value);
        traverseInOrder(node.right);
    }
}
```

If we call this method, the console output will show the in-order traversal: 3 4 5 6 7 8 9, where 6 is root and 4 left child and 8 right child of it

// Pre-order traversal visits first the **root node, then the left subtree, and finally the right subtree**:

```
public void traversePreOrder(Node node) {
    if (node != null) {
        System.out.print(" " + node.value);
        traversePreOrder(node.left);
        traversePreOrder(node.right);
    }
}
```

Output : 6 4 3 5 8 7 9

```
//  Post-order traversal visits the left subtree, the right subtree, and the root node at the end:
public void traversePostOrder(Node node) {
   if (node != null) {
      traversePostOrder(node.left);
      traversePostOrder(node.right);
      System.out.print(" " + node.value);
   }
}
```

Output : 3 5 4 7 9 8 6

```
//  2. Breadth-First Search
//  This type of traversal visits all the nodes of a level before going to the next level.
//  This kind of traversal is also called level-order and visits all the levels of the tree starting from the
    root, and from left to right. We'll use a Queue to hold the nodes from each level in order. We'll
    extract each node from the list, print its values, then add its children to the queue:

public void traverseLevelOrder() {
   if (root == null) {  return;    }

   Queue<Node> nodes = new LinkedList<>();
   nodes.add(root);
   while (!nodes.isEmpty()) {
      Node node = nodes.remove();
      System.out.print(" " + node.value);
      if (node.left != null) {   nodes.add(node.left);    }
      if (node.right!= null) {   nodes.add(node.right);   }
   }
}
```

LinkedList implements the **List**, **Queue** and **Deque** interfaces, as Deque extends the Queue interface.

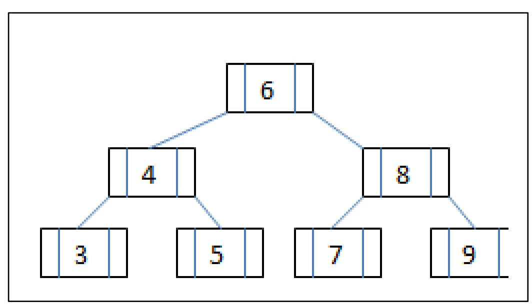
add an element to the end of the Queue
remove an element from the start

In this case, the order of the nodes will be: 6 4 8 3 5 7 9

```
}
```

File : **JavaApp1.java**

```
private static void driverBinaryTree()
   {
      //  Create a BinaryTree
      BinaryTree bt = new BinaryTree();
      bt.add(6);
      bt.add(4);
      bt.add(8);
      bt.add(3);
      bt.add(5);
      bt.add(7);
      bt.add(9);
      System.out.println(bt.root.value);
      System.out.println("Inorder Traversal : ");    bt.traverseInOrder(bt.root);
      System.out.println();
      System.out.println("Preorder Traversal : ");    bt.traversePreOrder(bt.root);
```



```
    System.out.println();
    System.out.println("Postorder Traversal : ");  bt.traversePostOrder(bt.root);
    System.out.println();
    System.out.println("Levelorder Traversal : "); bt.traverseLevelOrder();
    System.out.println();
    //  Find node 4
    Node fndNode = bt.containsNode(4);    PrintFoundNode(fndNode);
    fndNode = bt.containsNode(2);        PrintFoundNode(fndNode);

  }
  private static void PrintFoundNode(Node fndNode)
  {
    if (fndNode != null)
    {
      System.out.println("Found node : " + fndNode.value);
      System.out.println("Right of Found node : " + fndNode.right.value);
      System.out.println("Left of Found node : " + fndNode.left.value);
    }
    else
      System.out.println("node not found");
  }
```