

Model-View-Controller

We want to draw circles on a 2D area. The list of circles should be managed by the model. The user may add circles to the model by clicking on a free space in the area or remove circles when clicking at the position of a circle. The view displays the circles in the area. Use the MVC architecture to implement this application:

Model

- o class Circle: contains the x/y coordinates of the center of the circle and the radius and a method contains() that checks whether some coordinates are within an area of one circle
- o class CircleModel contains a list of Circles
- o class CircleEvent is the model (change) event of kind DELETED (when a circle is removed from the model) or ADDED (when a circle is added to the model)
- o interface CircleModelListener is the model listener

View (incl. Controller)

- o class CircleView is the view of the application – upon mouse click, new circles should be added or removed.

Class Circle is given as follows:

```
public class Circle {
private final int x; private final int y; private final int radius;

public int getX() { return x; }

public int getY() { return y; }

public int getRadius() { return radius;}

    public Circle(int x, int y, int radius) {
        this.x = x; this.y = y; this.radius = radius;
    }

    // returns true if x/y values are within the are of the circla
public boolean contains(int x, int y) {
double xDiff = Math.abs(this.x - x);
double yDiff = Math.abs(this.y - y);
return Math.sqrt(xDiff * xDiff + yDiff * yDiff) <= radius;

}

    @Override
    public String toString() {

return "Circle [x=" + x + ", y=" + y + ", radius=" + radius + "];" ;

}
}
```

Class CircleEvent is given as follows:

```
public class CircleEvent extends EventObject{ private static final long
serialVersionUID = 1L;

    public enum Kind { ADDED, DELETED };
    private final Circle circle;
    private final Kind kind;

public CircleEvent(CircleModel source, Circle circle, Kind kind) {
    super(source);

        this.circle = circle;
        this.kind = kind;
    }
public Kind getKind() { return kind; }

public Circle getCircle() { return circle;}

} }
```

The interface CircleModelListener is given as follows:

```
public interface CircleModelListener extends EventListener{ public void
circleEvent(CircleEvent event);

}}
```

Extend the class CircleModel with model listeners, adding and removing model listeners, and firing an event (note the in-code comments!).

```
public class CircleModel {
private List<Circle> circles = new ArrayList<>();

// Declare list of model listeners HERE:
private List<CircleModelListener> listeners = new ArrayList<>();

// Add model listeners HERE
public void addCircleModelListener(CircleModelListener listener) {

    listeners.add(listener);
}
// Remove model listener HERE
public void removeCircleModelListener(CircleModelListener listener) {

    listeners.remove(listener);
}
public List<Circle> getCircles(){
    return circles;
}

// ADD a new circle HERE in method void add (Circle circle)

public void add(Circle circle) {
circles.add(circle); fireModelEvent(CircleEvent.Kind.ADDED, circle);
}
```

```

}

// REMOVE a circle HERE

public void remove(Circle circle) {
    final boolean removed = circles.remove(circle); if (removed) {

        fireModelEvent(CircleEvent.Kind.DELETED, circle); }
}

// Implement the method fireModelChanged() HERE
// Parameters should be the CircleEvent.Kind and the circle

private void fireModelEvent(CircleEvent.Kind kind, Circle circle) {

    CircleEvent event = new CircleEvent(this, circle, kind); for (CircleModelListener
    listener : listeners) {

        listener.circleEvent(event); }
}

}

```

The view CircleView handles all mouse events to control adding / deleting Circles:

```

public class CircleView extends JComponent { ...

    private Color circleColor;
    private CircleModel model;

    public CircleView(CircleModel model, Color circleColor) { this.model =
    model;

    this.circleColor = circleColor;
    // Add a MouseListener, i.e., event listener to mouse event mouseReleased

    // Either REMOVE (if click on a circle) or ADD a circle
    addMouseListener(new MouseListener() {

@Override

        public void mouseReleased(MouseEvent e) {

            // REMOVE HERE

            for (Circle circle : model.getCircles()) {
                if (circle.contains(e.getX(), e.getY())) {

                    model.remove(circle);

                }
            }
        }
    }
}

```

```

}

// ADD HERE

model.add(new Circle(e.getX(), e.getY(), 50)); }

... });

// ADD model listener HERE; just repaint()

model.addCircleModelListener(new CircleModelListener() {

@Override
public void circleEvent(CircleEvent event) {

repaint(); } });
}

// Called when paint(), repaint() is called for the circle view @Override
protected void paintComponent(Graphics g) {

    super.paintComponent(g);
    Graphics g2d = g.create();
    g2d.setColor(circleColor);

// Clear all content in this circle view g2d.clearRect(0, 0, getWidth(),
getHeight());

// Draw each circle that is contained in the model / list of circles for
(Circle circle : model.getCircles()) {

    int diameter = circle.radius * 2;

g2d.fillOval(circle.x - circle.radius, circle.y - circle.radius, diameter,
diameter);

}}

```

Erstellen Sie unter Anwendung des MVC-Paradigmas eine Swing-Anwendung, die einen Punkt darstellen und verändern kann. Die Anwendung baut auf einem PointModel auf das einen Punkt (mit x/y-Koordinate) enthält. Der Punkt wird durch ein dargestellt. Mit einem Mausklick kann der Punkt auf die entsprechende Position versetzt werden. Des Weiteren gibt es ein Menu **Edit** mit einem Menüeintrag **Reset**, mit der der Punkt wieder auf die Ausgangsposition (X_Init/Y_Init) gesetzt wird.

Die Anwendung sieht in etwa folgendermaßen aus: (...)

Die Klasse **PointPanel** ist eine selbst gezeichnete Komponente, die von der Klasse **JPanel** abgeleitet ist. In der Methode **paintComponent** wird die aktuelle Position durch einen schwarzen Punkt angezeigt. Bei einem Mausklick im **PointPanel** soll der Punkt im Modell auf die Klickposition gesetzt werden. Das **PointPanel** muss dann neu gezeichnet werden.

Folgend finden Sie die Definitionen der Klassen, **Point**, **PointModel**, **PointListener**, **PointEvent**, **PointPanel** und **PointApp**. Ergänzen Sie die fehlenden Teile auf Basis der Kommentare.

```
public class Point {
    public final int x, y;

    public Point (int x, int y) {
        super(); this.x = x; this.y = y;
    }
}
```

```
import java.util.EventObject;
public class PointEvent extends Eventobject {
    private final Point point;

    public PointEvent(Object source, Point point) {
        super(source);
        this.point = point;
    }
    public Point getPoint () { return point; }
}
```

```
import java.util.EventListener;
public interface PointListener extends Eventlistener {
    // Event method which will be callend when point in model changes
    public void positionChanged(PositionEvent evt);
}
```

```

// Model for points
public class PointModel {
    // Field declarations (all fields must be private)
    private Point point;
    private final List<PointListener> listeners;

    public pointModel (int x, int y) {
        // initialize fields
        listeners = new ArrayList<PointListener>();
        this.point = new Point(x,y);
    }

    public Point getPoint() {
        return point;
    }

    public void changePoint(int x, int y) {
        // Set point with new coordinates x/y and signal change
        this.point = new Point(x,y);
        firePointChanged();
    }

    public void addPointListener (PointListener l) {
        // Add listener to listener list
        listener.add(l);
    }

    // Auxiliary method for notifying listeners
    public void firePointChanged() {
        PointEvent evt = new PointEvent(this, point);
        for (PointListener l : listener) {
            l.pointChanged(evt);
        }
    }
}

```

```

public class PointPanel extends JPanel {
    private final PointModel model;

    public PointPanel (PointModel model) {
        this.model = model;
        // Add listeners to event sources
        this.model.addPositionListener (pointListener);
        this.addMouseListener(clickHandler);
    }

    @Override
    protected void paintComponent (Graphics g) {
        super.paintComponent(g);
        int x, y;
        // Access data from model
        x = model.getXPos();
        y = model.getYPos();
        g.fillOval(x-3, y-3, 7, 7);
    }

    private final MouseListener clickHandler = new MouseAdapter () {
        // Implementation of mouse click events
        @Override
        public void mouseClicked(MouseEvent e) {
            model.changePoint(e.getX(), e.getY());
        }
    };

    private final PointListener pointListener = new PointListener() {
        // Implement the reaction to changes in model

```

```

        @Override
        public void pointChanged (PointEvent evt) {
            repaint();
        }
    };
}

```

```

public class PointApp {
    private static final int Y_INIT = 100, x_INIT = 100;

    private final JFrame frame;
    private final PointModel model;

    public PointApp (PointModel model) {
        this.frame = new JFrame ("Point");
        frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
        this.model = model;

        JMenuBar menuBar = new JMenuBar();
        frame.setJMenuBar(menuBar);
        JMenu editMenu = new JMenu("Edit");
        menuBar.add(editMenu);
        JMenuItem resetMenuItem = new JMenuItem("Reset");
        editMenu.add(resetMenuItem);
    }
    // Define action for resetMenuItem: reset point to coordinates X_INIT
    resetMenuItem.addActionListener(ae -> {
        this.model.changePoint(X_INIT, Y_INIT);
    });

    PointPanel panel = new PointPanel(model);
    Container cp = frame.getContentPane();
    cp.setLayout(new BorderLayout());
    cp.add(panel, BorderLayout.CENTER);
    frame.pack();
    frame.setVisible(true);
}

    public static void main (String[] args) {
        new PointApp (newPointModel(X_INIT, Y_INIT));
    }
}

```


Gegeben ist eine Klasse *Light*, die ein Licht repräsentiert. Das Licht kann man mit *turnOn* ein- und mit *turnOff* ausschalten. Die Methode *isOn* gibt an, ob das Licht ein- oder ausgeschaltet ist:

```
public class Light {
    private boolean on = false;
    public void turnOn() {
        on = true;
    }

    public void turnOff() {
        on = false;
    }

    public boolean isOn() {
        return on;
    }
}
```

Erweitern Sie diese Implementierung zu einem Model in einer Model-View-Controller (MVC) Anwendung. Definieren Sie ein Event-Objekt, ein Listener-Interface und erweitern Sie die Klasse *Light* entsprechend.

1.1 Implementierung Event-Objekt (3 Punkte)


Das Event-Objekt soll ein Feld für den aktuellen Zustand des Light-Objekts speichern.

Die Klasse muss von der Klasse *EventObject* abgeleitet werden. Beachten Sie dabei, dass die Klasse *EventObject* verlangt, dass im Konstruktor ein Source-Objekt initialisiert wird:

```
public class EventObject ... {
    public EventObject(Object source) { ... }
    ...
}
```

```
public class LightEvent extends EventObject {
    private boolean on;
```

csharp

 Copy code

```
    public LightEvent(Object source, boolean on) {
        super(source);
        this.on = on;
    }


    public boolean isOn() {
        return on;
    }
}
```

1.2 Definition des Listener-Interfaces

```
public interface LightListener {  
    public void lightTurnedOn(EventObject e);  
    public void lightTurnedOff(EventObject e);  
}
```

1.3 Erweiterung Klasse Light

csharp

 Copy code

```
public void turnOn() {  
    on = true;  
    notifyListeners();  
}  
  
public void turnOff() {  
    on = false;  
    notifyListeners();  
}  
  
public boolean isOn() {  
    return on;  
}  
  
public void addLightListener(LightListener listener) {  
    listeners.add(listener);  
}  
  
public void removeLightListener(LightListener listener) {  
    listeners.remove(listener);  
}  
  
private void notifyListeners() {  
    LightEvent event = new LightEvent(this, on);  
    for (LightListener listener : listeners) {  
        if (on) {  
            listener.lightTurnedOn(event);  
        } else {  
            listener.lightTurnedOff(event);  
        }  
    }  
}  
}
```

Implement a Swing application that allows to switch a light on (on is true) and off (on is false) when a user performs a mouse click on the window panel. The GUI looks as follows (left: off / right: on):

Based on the MVC (Model View Controller) architecture, the following interfaces and classes are used:

Model

- Class LightModel is the model for the light; it contains the light status (Boolean on) and methods to change the status of the light.
- Interface LightListener is a listener for changes of LightModel.
- Class LightEvent is the respective class for the light change event.

View/Controller

- Class LightViewController and class LightPanel are the classes responsible for visualizing the light status and observing user input (mouse click events). The application is already pre-programmed, fill-in code parts on how model changes are processed in the gap areas. Note the comments right before the areas.

```
public class LightModel {
    private boolean on;
    private List < LightListener > listeners = new ArrayList < LightListener > ();
    public boolean isOn() {
        return on;
    }
    public void changeLight() {
        this.on = !this.on;

        // TODO: Event has happened, signal this by calling the fire event method

        FireEvent();
    }

    public void addLightListener(LightListener l) {
        listeners.add(l);
    }
    public void removeLightListener(LightListener l) {
        listeners.remove(l);
    }

    // TODO: Implement the fire event method
    private void fireEvent() {
        LightEvent event = new LightEvent(this, on);
        for (LightListener listener: listeners) {
            listener.lightChanged(event);
        }
    }
}
```

```

public interface LightListener extends EventListener {
    public void lightChanged(LightEvent e);
}

```

```

Public class LightEvent extends Eventobject {
    private final boolean lightState;
    public LightEvent(Object source, boolean lightstate) {
        super(source);
        this.lightState = lightState;
    }
    public boolean isLightState() {
        return lightState;
    }
}

```

```

public class LightViewController {
    LightViewController(LightModel model) {
        JFrame frame = new JFrame("Light");
        JPanel panel = new LightPanel(model);
        frame.add(panel);
        frame.setVisible(true);
    }
}

```

```

public class LightPanel extends Janel {
    private final LightModel model;
    public LightPanel(LightModel model) {
        this.setBackground(Color... GRAY);
        this.model = model;
        this.addMouseListener(new MouseListener() {
            @Override
            public void mouseClicked(MouseEvent e) {

                // TODO: Change/switch the light
                model.changeLight();
            }
        });
    }
}

```

```

// Note, other mouse event listeners are not listed here due to saving space
});

```

```

// TODO: Add model listener and implement the action of the listener
model.addLightListener(new LightListener() {

```

```

    model.addLightListener(new LightListener() {
        @Override
        public void lightChanged(LightEvent e) {
            repaint();
        }
    });
    this.addMouseListener(new MouseListener()

```

```

    }

    @override protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        // TODO: If light is on, draw yellow circle

        If(model.isOn) {
            g.setColor(Color.YELLOW);
            g.fillOval(50, 50, 200, 200);
        }
        g.setColor(Color.DARK_GRAY); g.drawOval(50, 50, 200, 200);
    }
}

public class LightApp {
    public static void main(String[] args) {
        LightModel model = new LightModel();
        LightViewController viewControl = new LightViewController (model);
    }
}

```

Inheritance & Collections/Dynamic Binding

Vervollständigen Sie nachstehende Interfaces und Klassen zur Verwaltung von Visiten in Krankenhäusern entsprechend dem Visitor-Entwurfsmuster. Die abstrakte Klasse `Patient` deklarier eine Methode zum Entgegennehmen von Visitors vom Typ `PatientVisitor`. Die konkreten Klassen `CancerPatient` und `AccidentPatient` leiten von `Patient` ab. Das Interface `PatientVisitor` deklariert Methoden zum Besuchen von

`CancerPatient` und `AccidentPatient`. Die konkreten Visitor `Doctor` und `Nurse` implementieren `Interface PatientVisitor` und sollen einfache textuelle Aufgaben durchführen, z.B. "Doctor Frankenstein visits accident patient Meier" oder "Nurse Maria visits cancer patient Berger".

```
public abstract class Patient {
    private String name;
    public Patient (String name) { this.name = name; }
    public String getName() { return name; }
    // Method for accepting visitors
    public abstract void accept (PatientVisitor visitor);
}
```

```
public class CancerPatient extends Patient {
    public CancerPatient (String name) { super(name); }
    // Method for accepting visitors
    public void accept(PatientVisitor visitor) {
        visitor.visitCancer(this);
    }
}
```

```
public class AccidentPatient extends Patient {
    public AccidentPatient(String name) { super(name); }
    // Method for accepting visitors
    public void accept(PatientVisitor visitor) {
        visitor.visitAccident(this);
    }
}
```

```
public interface PatientVisitor {
    // Methods of visitor interface
    public void visitCancer (CancerPatient p);
    public void visitAccident (AccidentPatient p);
}
```

```
public class Doctor implements PatientVisitor {
    private String name;
    public Doctor(String name) {this.name = name; }
    public String getName() {return name; }

    // Implementation of visit methods

    public void visitCancer (CancerPatient p) {
        System.out.println("Nurse" + name + "visits" + "cancer patient" + p.getName());
    }
}
```

```

    public void visitAccident (AccidentPatient p) {
        System.out.println("Nurse" + name + "visits" + "accident patient" + p.getName());
    }
}

```

```

public class Nurse implements PatientVisitor {
    String name;
    public Nurse(String name) {this.name = name; }
    public String getName() {return name; }

    // Implementation of visit methods

    public void visitCancer (CancerPatient p) {
        System.out.println("Doctor" + name + "visits" + "cancer patient" + p.getName());
    }

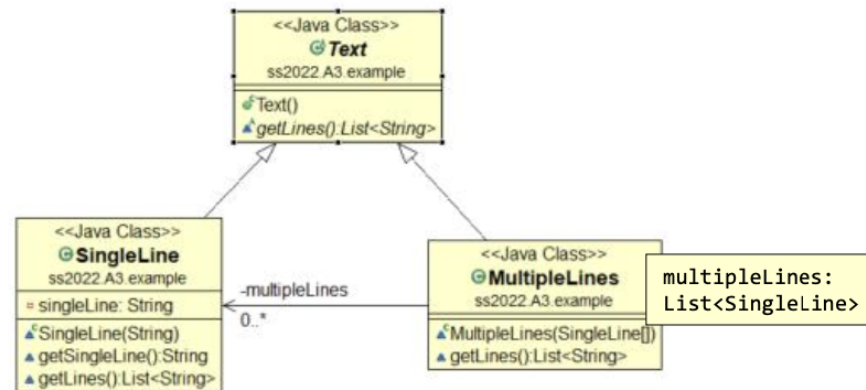
    public void visitAccident (AccidentPatient p) {
        System.out.println("Doctor" + name + "visits" + "accident patient" + p.getName());
    }
}

```

Anhang: Wesentliche Methoden von Stream, IntStream, Collectors, Optional

A3 – Inheritance and Dynamic Binding (14 Points)

The following class system allows to compose a text. Text is the abstract base class of the derived classes SingleLine (text with just one line) and MultipleLines consisting of a list of SingleLine text.



Text defines the abstract method `getLines()`, which returns a list of text Strings. **You should now implement the classes `SingleLine` and `MultipleLines` and all methods as defined in the UML class diagram.**

```
public abstract class Text {
    public abstract List<String> getLines();
}
```

class SingleLine: Implement a constructor, a getter method returning the line, and method `getLines()`

```
public class SingleLine extends Text {
    private final String line;

    public SingleLine(String line) {
        this.line = line;
    }

    public String getSingleLine(){
        return line;
    }

    @Override
    public List<String> getLines() {
        return List.of(line);
    }
}
```


class MultipleLines: Implement a **constructor** and method **getLines()**

```
public class MultipleLines extends Text {
    private final List<SingleLine> multipleLines;

    public MultipleLines(SingleLine...elems) {
        multipleLines = List.of(elems);
    }

    @Override
    public List<String> getLines() {
        return multipleLines.stream()
            .map( l -> l.getSingleLine())
            .collect (Collectors.toList());
    }
}
```

The code can be used to construct a message consisting of multiple single lines, as shown below.

```
public class Main {
    public static void main(String[] args) {
        Text prologueOfDune = new MultipleLines(
            new SingleLine("A beginning is a very delicate time."),
            new SingleLine("Know then, that it is the year 10191."),
            new SingleLine("The known universe is ruled by the Padisha Emperor Shaddam IV,
my father."),
            new SingleLine("In this time, the most precious substance in the Universe is
the spice melange."),
            new SingleLine("\n"),
            new SingleLine("[Princess Irulan]")
        );
    }
}
```

Gegeben ist ein Klassensystem für Drinks. *Drink* ist die Basisklasse, davon abgeleitet ist *SoftDrink* und *Alcoholics*. Jeder *Drink* hat einen Namen *name* und speichert einen Grundpreis *price* (Nettopreis).

```
public abstract class Drink {
    public final String name;
    public double price;
    protected Drink(String name, double price) {
        this.name = name;
        this.price = price;
    }
}

class SoftDrink extends Drink {
    public SoftDrink(String name, double price) { super(name, price); }
}

class Alcoholics extends Drink {
    public Alcoholics(String name, double price) { super(name, price); }
}
```

Gegeben ist des Weiteren die Klasse *Drinks*, die eine Liste von *Drink*-Objekten enthält. Die Klasse enthält auch bereits eine Methode *accept*, die die *accept*-Methoden der *Drink*-Objekte aufruft.


```
class Drinks {
    public final List<Drink> drinks;
    public Drinks(Drink...drinks) {
        this.drinks = List.of(drinks);
    }
    public void accept(DrinkVisitor v) {
        for (Drink drink: drinks) {
            drink.accept(v);
        }
    }
}
```

Implementieren Sie nach dem Visitor-Pattern einen *DrinkVisitor* für *Drink*-Objekte wie folgt: **2.1**

DrinkVisitor (3 Punkte)

Definieren Sie ein Interface *DrinkVisitor* mit Methoden für die unterschiedlichen *Drink*-Typen.

java

 Copy code

```
interface DrinkVisitor {
    void visitSoftDrink(SoftDrink soft);
    void visitAlcoholics(Alcoholics beer);
}
```

2.2 Accept-Methoden (3 Punkte)

Implementieren Sie *accept*-Methoden in den *Drink*-Klassen zum Akzeptieren von *DrinkVisitors*.
Anmerkung: Sie müssen nur die Methoden *accept* schreiben.

Methode *accept* in Klasse *Drink*

```
public abstract class Drink {  
    public final String name;  
    public double price;  
    protected Drink(String name, double price) {  
        this.name = name;  
        this.price = price;  
    }  
    public abstract void accept(DrinkVisitor v);  
}
```

Methode *accept* in Klasse *SoftDrink*

```
class SoftDrink extends Drink {  
    public SoftDrink(String name, double price) {  
        super(name, price);  
    }  
    public void accept(DrinkVisitor v) {  
        v.visitSoftDrink(this);  
    }  
}
```

Methode *accept* in Klasse *Alcoholics*

```
class Alcoholics extends Drink {  
    public Alcoholics(String name, double price) {  
        super(name, price);  
    }  
    public void accept(DrinkVisitor v) {  
        v.visitAlcoholics(this);  
    }  
}
```

2.3 TaxVisitor (4 Punkte)

Implementieren Sie eine konkrete Klasse *TaxVisitor* als Implementierung von *DrinkVisitor*, der die Steuer für die unterschiedlichen Drinks berechnet und in einer Variablen *tax* aufsummiert.

```
class TaxVisitor implements DrinkVisitor {  
    public double tax;  
  
    ... }  
}
```

Dabei berechnet sich die Steuer unterschiedlich für die unterschiedlichen Typen von *Drinks* wie folgt:

- *SoftDrink* : 10% Umsatzsteuer vom Nettopreis
- *Alcoholics* : 20% Umsatzsteuer vom Nettopreis

Klasse *TaxVisitor*

```
typescript Copy code  
  
class TaxVisitor implements DrinkVisitor {  
    public double tax;  
  
    public void visitSoftDrink(SoftDrink soft) {  
        tax += soft.price * 0.1;  
    }  
    public void visitAlcoholics(Alcoholics beer) {  
        tax += beer.price * 0.2;  
    }  
}
```

Inheritance and Collections (30 Points) 7

Assume that you want to implement a management software for European Championship soccer teams

("EM Fußballmannschaften"). A soccer team (class Team) consists of a sorted set of players (class Player) and is from a specific country. A player has a first name and a lastname, as well as a position in the game (enum Position) and a tricot number. Use the Java Collections API and add the missing code parts in the following to implement the soccer team classes and the test class.

```
public enum Position {
    GOAL,
    DEFENSE,
    MIDFIELD,
    FORWARD;
}

public class Player implements Comparable< Player > {
    private final String firstName;
    private final String lastName;
    private Position position;
    private final int tricotNumber;

    public Player(String firstName, String lastName, Position position, in tricotNumber)
    {
        this.firstName = firstName;
        this.lastName = lastName;
        this.position = position;
        this.tricotNumber = tricotNumber;
    }

    @override
    public int compareTo(Player other) {
        int compare = this.position.ordinal() - other.position.ordinal();
        if (compare == 0) {
            compare = this.lastName.compareTo(other.lastName);
        }
        if (compare == 0) {
            compare = this.firstName.compareTo(other.firstName);
        }
        return compare;
    }

    @override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obi == null) return false;
        if (getClass() != obj.getClass()) return false;

        // TODO: Implement equality of this and obj by using all member fields
        Player other = (Player) obj;
        if (!firstName.equals(other.firstName)) return false;
        if (!lastName.equals(other.lastName)) return false;
    }
}
```

```
    if (position != other.position) return false;
    if (tricotNumber != other.tricotNumber) return false;
```

```
}
```

```
@Override
```

```
public int hashCode() { ...
```

```
}
```

```
@Override
```

```
String toString() { ...
```

```
}
```

```
public String getFirstName() {
```

```
    return firstName;
```

```
}
```

```
public String getLastName() {
```

```
    return lastName;
```

```
}
```

```
public Position getPosition() {
```

```
    return position;
```

```
}
```

```
public void setPosition(Position position) {
```

```
    this.position = position;
```

```
}
```

```
public int getTricotNumber() {
```

```
    return tricotNumber;
```

```
}
```

```
}
```

```
public enum Country {
```

```
    AUSTRIA,
```

```
    GERMANY,
```

```
    ENGLAND,
```

```
    FRANCE,
```

```
    IRELAND,
```

```
    SPAIN,
```

```
    ITALY,
```

```
    POLAND;
```

```
}
```

```
public class Team {
```

```
    private final Country country;
```

```
// TODO: Define a fitting sorted collection of players of the team
```

```
SortedSet < Player > players = new TreeSet < > ();
```

```
//TODO: Initialize country and the collection of players
```

```
public Team(Country country) {
```

```
    this.country = country;
```

```
    this.players = new TreeSet < > ();
```

```

    }

    //TODO: Add a player to the team
    public void addTeamMember(Player player) {
        this.players.add(player);

    }

    //TODO: Remove a player from the team
    public void removeTeamMember(Player player) {
        players.remove(player);

    }

    //TODO: Return the player with a specific tricotNumber, or null if non existing
    public Player getTeamMember(int tricotNumber) {
        for (Player player: players) {
            if (player.getTricotNumber() == tricotNumber) {
                return player;
            }
        }
        return null;

    }

    // print out team members
    public void printLineUp() {
        System.out.println("\nLine-Up of " + country + ":");
        for (Player p: players) {
            System.out.println(p.toString());
        }
    }
}

public class Test {
    public static void main(String[] args) {
        Team austria = new Team(Country.AUSTRIA);
        austria.addTeamMember(new Player("Robert", "Almer", Position.GOAL, 1));
        austria.addTeamMember(new Player("Aleks", "Dragovic", Position.DEFENSE,
3));
        austria.addTeamMember(new Player("Christian", "Fuchs", Position.DEFENSE,
5));
        austria.addTeamMember(new Player("Florian", "Klein", Position.DEFENSE,
17));
        austria.addTeamMember(new Player("Sebastian", "Proedl", Position.DEFENSE,
15));
    }
}

```

```

    austria.addTeamMember(new Player("Zlatko", "Junuzovic", Position.DEFENSE,
10));
    austria.addTeamMember(new Player("David"
        "Alaba", Position.MIDFIELD, 8)),
        austria.addTeamMember(new Player("Julian"
            "Baumgartlinger", Position.MIDFIELD, 14));
    austria.addTeamMember(new Player("Marko"
        "Arnautovic", Position.MIDFIELD, 7));
    austria.addTeamMember(new Player("Martin", "Harnik", Position.MIDFIELD,
11));
    austria.addTeamMember(new Player("Marc",
        "Janko", Position.FORWARD, 21));
    // TODO: Change position of player with number 8 (David Alaba) to FORWARD
    Player player = austria.getPlayerByTricotNumber(8);
    if (player != null) {
        player.setPosition(Position.FORWARD);

        austria.printLineUp();
    }
}

```

Expected output of test program will, e.g., look like:

Line - Up of AUSTRIA:

```

Player[Robert Almer plays at GOAL with tricotNumber 1]
Player[Aleks Dragovic plays at DEFENSE with tricotNumber 3]
Plaver[Christian Fuchs plays at DEFENSE with tricotNumber 5]
Player[Zlatko Junuzovic plays at DEFENSE with tricotNumber 10]
Player[Florian Klein plays at DEFENSE with tricotNumber 17]
Player[Sebastian Proedl plays at DEFENSE with tricotNumber 15]
Player[Marko Arnautovic plays at MIDFIELD with tricotNumber 7]
Player[Julian Baumgartlinger plays at MIDFIELD with tricotNumber 14]
Player[Martin Harnik plays at MIDFIELD with tricotNumber 11]
Player[David Alaba plays at FORWARD with tricotNumber 8]
Player[Marc Janko plays at FORWARD with tricotNumber 21]

```


STREAMS

The class Person is given as follows:

```
public class Person {  
    private String name; private int age; private Set<String> hobbies;  
  
    public Person(String name, int age, Set<String> hobbies){  
        this.name = name;  
        this.age = age;  
        this.hobbies = hobbies;  
    }  
  
    String getName() { return name; }  
    int getAge() { return age; }  
    Set<String> getHobbies() { return hobbies; }  
}
```

A list containing persons is given as follows:

```
List<Person> persons = List.of(  
    new Person ("Hans",42,Set.of("Dancing","Reading","Travelling")),  
    new Person ("Toni",17,Set.of("Gaming","Guitar Playing")),  
    new Person ("Klara",22,Set.of("Skiing","Sleeping")));
```

Use the following STREAM OPERATIONS to answer the questions below:

- Stream<T> filter(Predicate<? super T> predicate)
- Stream<T> sorted()
- <R> Stream<R> map(Function<? super T, ? extends R> mapper)
- <R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)
- Optional<T> findFirst()
- long count()
- <R, A> R collect(Collector<? super T, A, R> collector)
 - To create a list: static Collectors<T, ?, List<T>> toList()

A) Calculate the number of persons younger than 18 (hint: use filter and count):

2 Points

long numPersonsUnder18 =

```
long count = persons.stream()  
    .filter(person -> person.getAge() < 18)  
    .count();
```

B) Generate a list of person names with hobby Reading, the list should be sorted alphabetically (natural order; hint: use filter, map, sorted, and collect):

4 Points

List<String> namesSorted =

```
List<String> namesSorted = persons.stream()  
    .filter(person -> person.getHobbies().contains("Reading"))  
    .map(Person::getName) .sorted() .collect(Collectors.toList());  
}
```

C) Check, whether there exists the hobby Skiing among persons (hint: use flatMap, filter, findFirst - and note that the method findFirst returns an Optional):

4 Points

```
boolean exists = persons.stream()  
    .anyMatch(person -> person.getHobbies().contains("Skiing"));
```

B – Streams (10 Points)

The following list of words is given:

```
List<String> words =  
List.of("This", "list", "contains", "the", "movies", "Titanic", "and", "Avatar");
```

Implement the following operation by **USING THE STREAM API!**

Operation 1 (4 Points):

Use stream operations to filter words that start with an uppercase letter, map them to lower case words and create/return a list containing these words.

Hints:

- The method `boolean Character.isUpperCase(char c)` returns true if the parameter is an upper case letter, else it returns false; the String method `toLowerCase()` can be used to change a String into lower case only characters.
- The following stream operations can be used to filter, map, and return the required list:
 - `Stream<T> filter(Predicate<? super T> predicate)`
 - `<R> Stream<R> map(Function<? super T,? extends R> mapper)`
 - `<R, A> R collect(Collector<? super T, A, R> collector)` in combination with `Collector<T, ?, List<T>> toList()` of class `Collectors`.

```
List<String> toLowerCaseWords =  
    words.stream()  
        .filter(w -> Character.isUpperCase(w.charAt(0)))  
        .map(w->w.toLowerCase())  
        .collect(Collectors.toList());
```

Operation 2 (3 Points):

Use stream operations to order the words alphabetically and to print them out.

Hints: Use the following stream operations

- `Stream<T> sorted()`
- `void forEach(Consumer<? super T> action)`

```
words.stream()  
    .sorted()  
    .forEach(System.out::println);
```

Operation 3 (3 Points):

Use stream operations to print the words with a space between the words.

```
// one possible solution; one can also use collect(Collectors.joining(" "))  
words.stream()  
    .forEach(s -> System.out.print(s + " "));
```

High Order Function

A 4 – Higher Order Function (10 Points)

Define a method `doForEach(Function<String,String> action)` that allows to implement a transforming action for each `String` element of a class `MyList`. `MyList` should implement the interface `DoForEach`.

The functional interface `Function` is defined as follows:

```
@FunctionalInterface
public interface Function<T,R> {
    R apply(T t);
}
```

The interface `DoForEach` is defined as follows:

```
public interface DoForEach{
    public void doForEach(Function<String,String> action);
}
```

Implement the method `doForEach` in class `MyList`:

```
public class MyList implements DoForEach {
    final List<String> myList;

    MyList (String...elems){
        myList = List.of(elems);
    }

    // TODO: Implement method doForEach here
}
```

```
public void doForEach(Function<String, String> action) { for (String s : myList)
{
    String result = action.apply(s);
    System.out.println(result); }
}
```

The list is filled as follows:

```
MyList myList =
    new MyList("Star Trek: Picard","Big Bang Theory","The Queen's Gambit");
```

Now, use the method `doForEach` to turn each character into an upper case character (Hint: Use method: `String toUpperCase()`; available for `String` objects):

```
myList.doForEach(s -> s.toUpperCase());
```

A - HOF (Higher Order Functions) – 10 Points:

Extend the Text example in Task A 3 by adding a new HOF doForEach() that implements a test and a consuming user for each line. The functional interfaces Predicate and Consumer are given as follows:

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
}
```

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
}
```

The method is defined as an abstract method in abstract class Text:

```
public abstract class Text {
    ...
    public abstract doForEach(Predicate<String> pred, Consumer<String> user);
}
```

Add an implementation of the new method in class SingleLine

```
public class SingleLine extends Text {
    ...
    @Override
    public void doForEach(Predicate<String> pred, Consumer<String> user) {
        if (pred.test(line)) user.accept(line);
    }
}
```

Add an implementation of the new method in class MultipleLines

```
public class MultipleLines extends Text {
    ...
    @Override
    public void doForEach(Predicate<String> pred, Consumer<String> user) {
        for (SingleLine l : multipleLines) {
            l.doForEach(pred, user);
        }
    }
}
```

Use the method doForEach() to filter lines with more than 20 characters and printout the remaining prologue lines of Dune of prologueOfDune (see Task A3):

```
prologueOfDune.doForEach(l->l.length() > 1, System.out::println);
```

Wahr/Falsch

1. A class may extend multiple classes. **FALSE**

A class in Java may only extend one class, it cannot extend multiple classes. Java does not support multiple inheritance of classes, but it does support multiple inheritance of interfaces.

2. An abstract class contains only abstract methods. **FALSE**

This statement is not entirely true. An abstract class can contain both abstract and non-abstract methods. An abstract method is a method that is declared without an implementation (i.e., no method body), while a non-abstract method is a method that has a method body. An abstract class is a class that cannot be instantiated and is meant to be subclassed. It's often used to provide a common base class implementation for subclasses.

3. When a member field is declared private, it can only be accessed inside the class. **TRUE**

4. Class ClassA is given as follows:

```
class ClassA {
```

```
    public static void methodA() {  
        System.out.println("SE2 is awesome.");  
    }
```

A ClassA object is created as follows:

The method of ClassA may be called as: ClassA. methodA(). **TRUE**

5. Class Person, class Student are given as follows:

```
class Person {
```

```
    private final String firstName;  
    Person (String firstName) { this.firstName = firstName; }
```

```
class Student extends Person {
```

```
    Student (String firstName) { super (firstName); }
```

```
    public void doStudy () { System.out.println("Studying SE2"); } }
```

```
Person p = new Student ("Peter");
```

The following method call works: p.doStudy () ; **FALSE**

*This statement is false because the reference variable **p** is of type **Person** and the **Person** class does not have a **doStudy** method. Although **p** refers to an instance of **Student**, the type of the reference variable determines which methods can be*

called on the object. Since `p` is of type `Person`, only the methods that are declared in the `Person` class or its superclasses are accessible. The `doStudy` method is a member of the `Student` class and is not accessible through the `Person` reference.

6. A `SortedSet` allows duplicates. **FALSE**

This statement is false because a `SortedSet` does not allow duplicates.

7. Every terminal stream operation returns a `Stream`. **FALSE**

This statement is false because not every terminal stream operation returns a `Stream`. Terminal operations are operations that produce a non-stream result or a side-effect, and they signal the end of a stream pipeline. Examples of terminal operations include `forEach`, `count`, `min`, `max`, `reduce`, `collect`, etc. Most terminal operations return a non-stream result such as a scalar value, a collection, or `void`. Only a few terminal operations, such as `flatMap`, return a `Stream`. Therefore, it's incorrect to say that every terminal stream operation returns a `Stream`.

8. Predicate from `java.util.function` is used in this example. The method `public void function (Predicate p)` is a higher order function (HOF). **TRUE**

9. With the following code, a new `Thread` is started.

```
Thread t = new Thread( () ->System.out.println( "Running ..."));
```

FALSE

This statement is false because the code creates a new `Thread` object, but it does not start the thread. The `Thread` constructor takes a `Runnable` object as an argument, and this argument specifies the code to be executed by the new thread. However, calling the `Thread` constructor does not start the thread. To start the thread, you need to call the `start` method on the `Thread` object, for example:

```
Thread t = new Thread( () -> System.out.println( "Running ..."));  
t.start();
```

10. With the Model-View-Controller architecture, it is possible to create multiple controllers without changing the model. **TRUE**