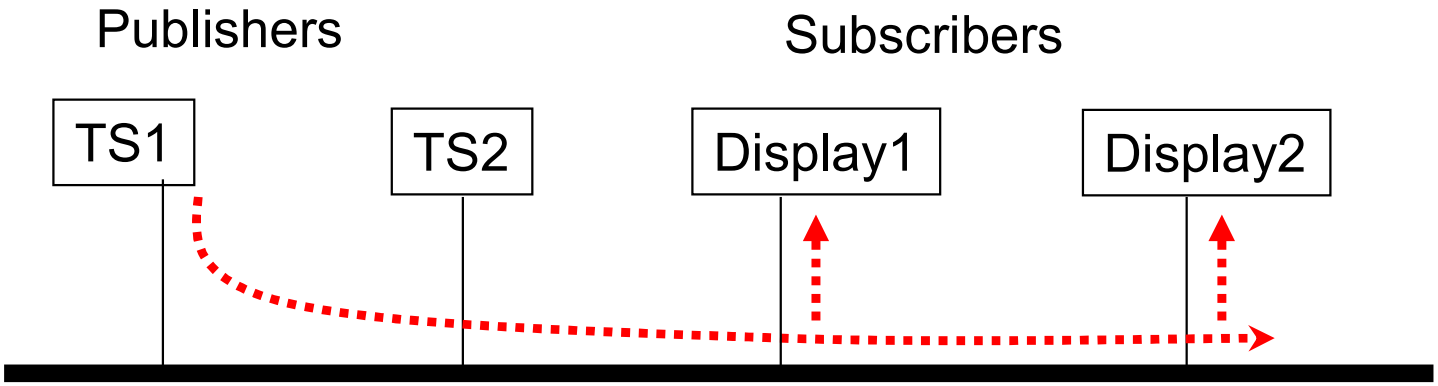
Assignment 1 Bonus point

or

Assignment 2 standard variant A4

Requirements details

Example: Multiple Sensors and Multiple Displays



Publishers: various sensors generating events of type TemperatureEvent and various water level monitors generating events of type WaterLevelMonitor.
Subscribers: various displays. Displays subscribe to event types and receive notification from all publishers that produce these events. If a display wants to receive events (TemperatureEvent) only from a part of the publishers, it can add Filters to its subscription.

Example with the EventNotifier

```
import EventService.EventService;

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

        GeneralDisplay d1 = new GeneralDisplay("Display1");
        GeneralDisplay d2 = new GeneralDisplay("Display2");

        EventService.instance().subscribe(TemperatureEvent.class, null, d1);

        EventService.instance().subscribe(TemperatureEvent.class, new LocationFilter("Arad"), d2);


        TemperatureSensor ts1 = new TemperatureSensor("TS001", "Arad", 99.9F);
        TemperatureSensor ts2 = new TemperatureSensor("TS002", "Timisoara", 98.5F);

        ts1.setTemperatureValue(24);
        ts2.setTemperatureValue(12);
        ts1.setTemperatureValue(5);
        ts2.setTemperatureValue(8);

    }
```

Events with the EventNotifier

```
public class TemperatureEvent implements EventService.Event{
    TemperatureSensor theSensor;
    public TemperatureEvent(TemperatureSensor t){
        theSensor=t;
    }
    public TemperatureSensor getTheSensor(){
        return theSensor;
    }
}

public class WaterLevelEvent implements EventService.Event{
    WaterLevelMonitor theMonitor;
    public WaterLevelEvent(WaterLevelMonitor t){
        theMonitor=t;
    }
    public WaterLevelMonitor getTheMonitor(){
        return theMonitor;
    }
}
```

A Publisher with the EventNotifier

```
import EventService.EventService;

public class TemperatureSensor {
    private int temperatureValue;
    private String ID;
    private String location;
    private float precision;

    public TemperatureSensor(String ID, String location, float precision) {
        this.ID=ID;
        this.location=location;
        this.precision=precision;
        this.temperatureValue=0;
    }

    public void setTemperatureValue(int newValue){
        temperatureValue=newValue;
        EventService.instance().publish(new TemperatureEvent(this));
    }

// …
```

A Subscriber with the EventNotifier

```
public class GeneralDisplay implements Subscriber {
    int value;
    String explanation;
    String name;

// ….

    @Override
    public void inform(Event event) {
        if (event instanceof TemperatureEvent) {
            TemperatureEvent e = (TemperatureEvent) event;
            value = e.getTheSensor().getTemperatureValue();
            explanation = " the temperature in " + e.getTheSensor().getLocation();
        }
        if (event instanceof WaterLevelEvent) {
            WaterLevelEvent e = (WaterLevelEvent) event;
            WaterLevelMonitor m = e.getTheMonitor();
            value = m.getLevelValue();
            explanation = " the water height on river " + m.getRiver() + " at km " + m.getPosition();
        }
        display();
    }
}
```

Problem: all Subscribers must have the same handler method, imposed by the Subscriber interface, and thus they cannot handle different event types in a “nicer” way

Requirements for the Improved EventBus

- No common type for Events, each should be free to define own event types
- Each Subscriber must be free to define its own kinds of handler methods, having as arguments its own subscribed Event types, thus no more fixed Subscriber interface
  - **Solution Variant 1: The subscriber transmits the name of its handler method when subscribing to an event type**
  - **Solution Variant 2: The subscriber uses custom annotations to signalize which ones of its methods are event handlers**
  - **Both solution variants rely on using reflective languages (see Lecture week 4)**

Example: The Subscriber in Solution Variant 1

```
public class GeneralDisplay {
    int value;
    String explanation;
    public GeneralDisplay() {
        value=0;
        explanation="";
    }
    public void onTemperatureEvent(NewTemperatureEvent e){
          value= e.getTheSensor().getTemperatureValue();
          explanation=" the temperature in "+e.getTheSensor().getLocation();
          display();
        }

    public void onWaterlevelEvent(NewWaterlevelEvent e){
        WaterLevelMonitor m=e.getTheMonitor();
        value= m.getLevelValue();
        explanation=" the water height on river "+m.getRiver()+" at km "+m.getPosition();
        display();
    }

        public void display(){
        System.out.println(value+explanation);
        }
}
```

Example: How to subscribe in Solution Variant 1

```
import NewEventBus.*;

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

        GeneralDisplay d1 = new GeneralDisplay("Display1");
        GeneralDisplay d2 = new GeneralDisplay("Display2");

        EventBus.instance().subscribe(d1, “onTemperatureEvent”);

        EventBus.instance().subscribe(d1, “onWaterLevelEvent”);

// …

    }
```

The name of the handler function is given at subscription time.
It must be an existing method of the subscriber, having one argument which is then by default the subscribed event type

Example: How to subscribe in Solution Variant 1

```
import NewEventBus.*;

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

      GeneralDisplay d1 = new GeneralDisplay("Display1");
      GeneralDisplay d2 = new GeneralDisplay("Display2");

      EventBus.instance().subscribe(d1, “onTemperatureE

      EventBus.instance().subscribe(d1, “on

// … various sensors

   }
```

The implementation of the New EventBus has a different **subscribe** method: it receives an object which will be the subscriber and the name of its handler. The EventBus will check (***using Reflection***) that the class of the subscriber object contains such a method and that it has one argument, and considers the type of the argument the subscribded event type. If a subscription is invalid, throw exception

Example: Adding filters in Solution Variant 1

```
import NewEventBus.*;

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

        GeneralDisplay d1 = new GeneralDisplay("Display1");
        GeneralDisplay d2 = new GeneralDisplay("Display2");

        EventBus.instance().subscribe(d1, “onTemperatureEvent”, new LocationFilter(“Arad”), “checkLocation”);

// …

    }
```

Specifying filter methods is done in a similar way as handler methods, by their name. The EventBus must check that the class of the provided filter object has a method with the given name, returning a boolean and having one argument of the same type as the handler method

Example: The Subscriber in Solution Variant 2

```
public class GeneralDisplay {
   int value;
   String explanation;
   public GeneralDisplay() {
      value=0;
      explanation="";
   }


 @ThisIsMyHandler
   public void onTemperatureEvent(NewTemperatureEvent e){
         value= e.getTheSensor().getTemperatureValue();
         explanation=" the temperature in "+e.getTheSensor().getLocation();
         display();
      }

 @ThisIsMyHandler
   public void onWaterlevelEvent(NewWaterlevelEvent e){
      WaterLevelMonitor m=e.getTheMonitor();
      value= m.getLevelValue();
      explanation=" the water height on river "+m.getRiver()+" at km "+m.getPosition();
      display();
   }
```

Example: How to subscribe in Solution Variant 2

```
import NewEventBus.*;

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

        GeneralDisplay d1 = new GeneralDisplay("Display1");
        GeneralDisplay d2 = new GeneralDisplay("Display2");

        EventBus.instance().subscribe(d1);

// … various sensors

    }
```

The implementation of the New EventBus has a different **subscribe** method: it is enough to receive an object which will be the subscriber. The EventBus will check the class of the subscriber object for methods that contain the custom annotation and it will identify in this way the subscribed event types and their handlers. If a subscription is invalid, throw exception

Example: Adding filters in Solution Variant 2

```
import NewEventBus.*;

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

        GeneralDisplay d1 = new GeneralDisplay("Display1");
        GeneralDisplay d2 = new GeneralDisplay("Display2");

        EventBus.instance().subscribe(d1,  new LocationFilter("Arad"));

// …

    }
```

Subscribe only with subscriber object and filter object. The filter methods are specified with annotations in the filter class. The EventBus must check that the class of the provided filter object has an annotated method, returning a boolean and having one argument of the same type as the handler method

Example: defining custom annotations

```
import java.lang.annotation.Annotation;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.reflect.Method;

//define the custom annotation ThisIsMyHandlerFunction
//the annotation must be present at runtime
@Retention(RetentionPolicy.RUNTIME)
@interface ThisIsMyHandlerFunction {
}
```