Using Publish-Subscribe Infrastructures

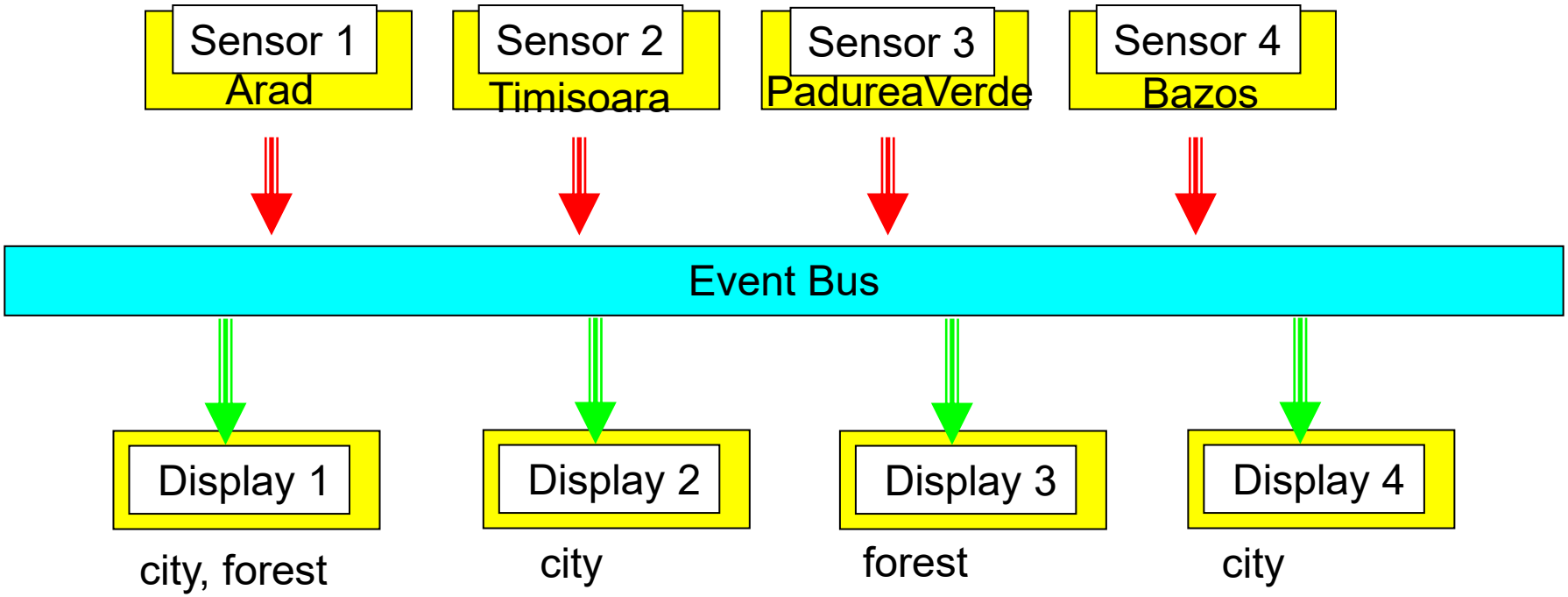
Case study: Greenrobot

Event Service Infrastructures

- **Infrastructures** that provide a Publish-Subscribe API to be used by applications; Examples:
- **In-process EventBus**:
  - Greenrobot eventbus
  - Akka event bus
  - Alternative: ReactiveX - RXJava
- **Inter-process, distributed Publish-Subscribe** infrastructure: part of middleware for distributed computing
  - Message Broker vs. EventBus
  - RabbitMQ, ActiveMQ, Kafka
  - JMS

Motivating Example: A Scenario with Sensors and Displays

- A variable number of Sensors periodically report measured temperature values
- A variable number of displays show values from sensors
- Sensors may be in a city or in a forest. Each display should be configurable to follow only city temps, only forest temps, or both.



Example: Greenrobot - an in-process EventBus for Java

https://greenrobot.org/eventbus/documentation/how-to-get-started/

GreenRobot EventBus

- **The EventBus:** `org.greenrobot.eventbus.EventBus;`
- **Events:** POJO (plain old Java objects) without any specific requirements
- **Publishers:** any object can be a publisher by calling `EventBus.post(event);`
- **Subscribers:** an object registers as a subscriber by calling `EventBus.register(subscriber);`
  - Subscribers class must implement event handling methods (also called “subscriber methods”) that will be called when an event is posted. These are defined with the `@Subscribe` annotation.
  - By default the subscribed event type is the type of this method’s argument (the subscribed event is inferred from the type of this method’s argument)

Example with Greenrobot

```
 public enum EnvironmentType {
      CITY,
      FOREST
  }


public class TemperatureEvent {
    private final String sensorId;
    private final String location;
    private final float value;
    private final EnvironmentType environmentType;

    public TemperatureEvent(String sensorId,
                            String location,
                            float value,
                            EnvironmentType environmentType) {
      // … details omitted on slide
  }

}
```

```java
import org.greenrobot.eventbus.EventBus;

public class TemperatureSensor {   // a publisher

    private final EventBus eventBus;
    private final String id;
    private final String location;
    private final EnvironmentType environmentType;
    private float temperatureValue;

    public TemperatureSensor(EventBus eventBus,
                             String id,
                             String location,
                             EnvironmentType environmentType,
                             float temperatureValue) {
     // …details omitted on slide
    }

    public void setTemperatureValue(float tValue) {
        this.temperatureValue = tValue;

        TemperatureEvent event =
            new TemperatureEvent(id, location, tValue, environmentType);

        eventBus.post(event);
    }
}
```

```java
import org.greenrobot.eventbus.Subscribe;

public class GeneralDisplay { //a subscriber

    private final String displayName;
    private final Set<EnvironmentType> followedTypes;

    public GeneralDisplay(String displayName,
                          Set<EnvironmentType> followedTypes) {
        this.displayName = displayName;
        this.followedTypes = followedTypes;
    }

    @Subscribe
    public void onTemperatureEvent(TemperatureEvent event) {

        if (!followedTypes.contains(event.getEnvironmentType())) {
            return; // ignore this event
        }

        System.out.println(displayName + " received from Sensor "
                + event.getSensorId()
                + " =" + event.getValue());
    }
}
```

```
public class MainSensors {

    public static void main(String[] args) {

        EventBus eventBus = EventBus.getDefault();

        GeneralDisplay d1 = new GeneralDisplay("Display1",
                Set.of(EnvironmentType.CITY, EnvironmentType.FOREST));
        eventBus.register(d1);

         // ... similar for GeneralDisplay d2, d3, d4


        TemperatureSensor ts1 = new TemperatureSensor(eventBus,
                "TS001", "Arad", EnvironmentType.CITY, 0);

        // ... similar for TemperatureSensor ts2, ts3, ts4

        // simulate sensors start working
        ts1.setTemperatureValue(25);
        ts2.setTemperatureValue(10);
        ts3.setTemperatureValue(25);
        ts4.setTemperatureValue(10);

    }
```

Design Decisions: The Event Types

- One single event type, additional event data filtered by subscribers:

```
class TemperatureEvent {…}

@Subscribe
public void onTemperatureEvent(TemperatureEvent event) {
        if (followedTypes.contains(event.getEnvironmentType())) …
}
```

- Multiple event types, multiple handler functions:

```
class CityTemperatureEvent {…}
class ForestTemperatureEvent {…}

@Subscribe
public void onCityTemperatureEvent(CityTemperatureEvent event) {
…
}
@Subscribe
public void onForestTemperatureEvent(ForestTemperatureEvent event) {
…
}
```

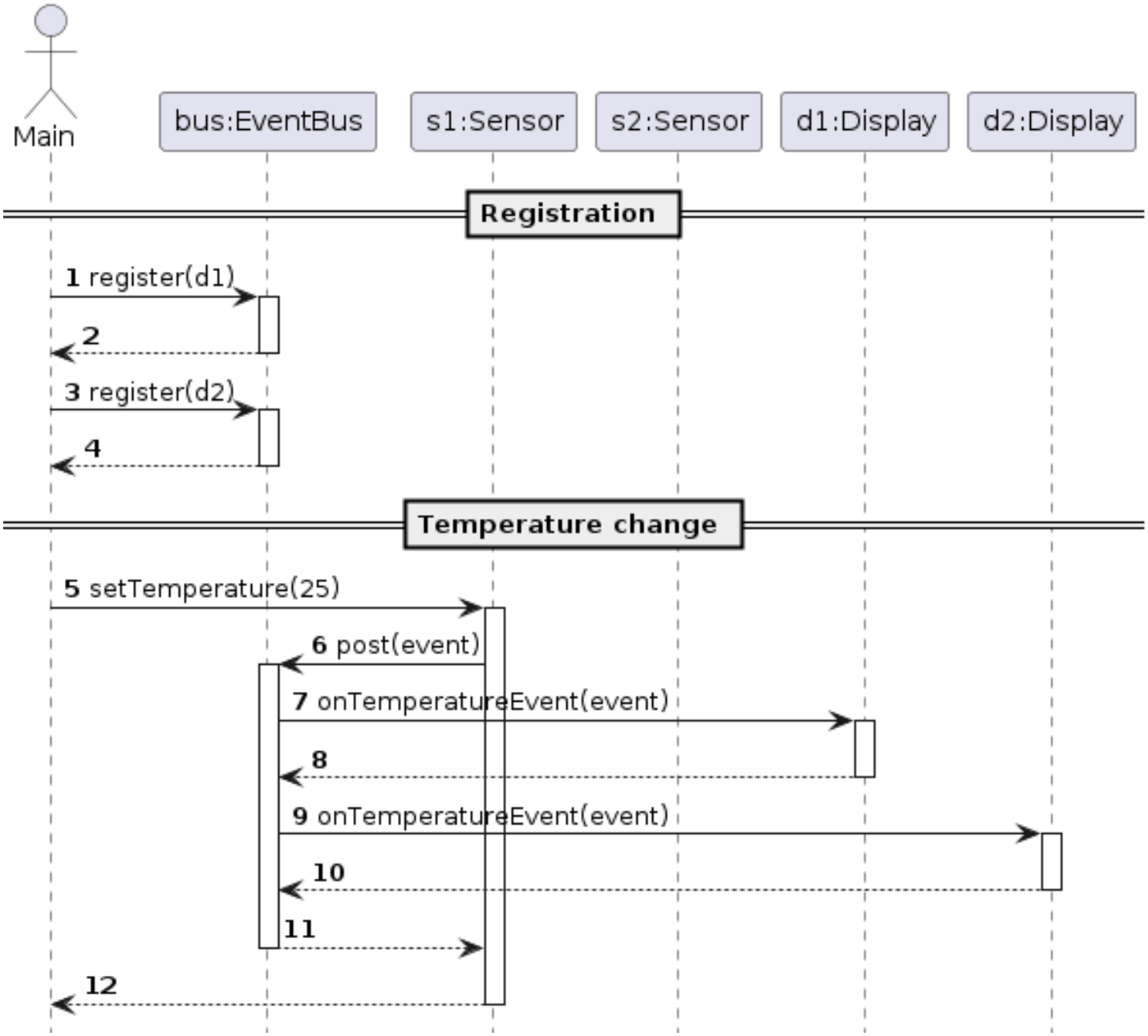
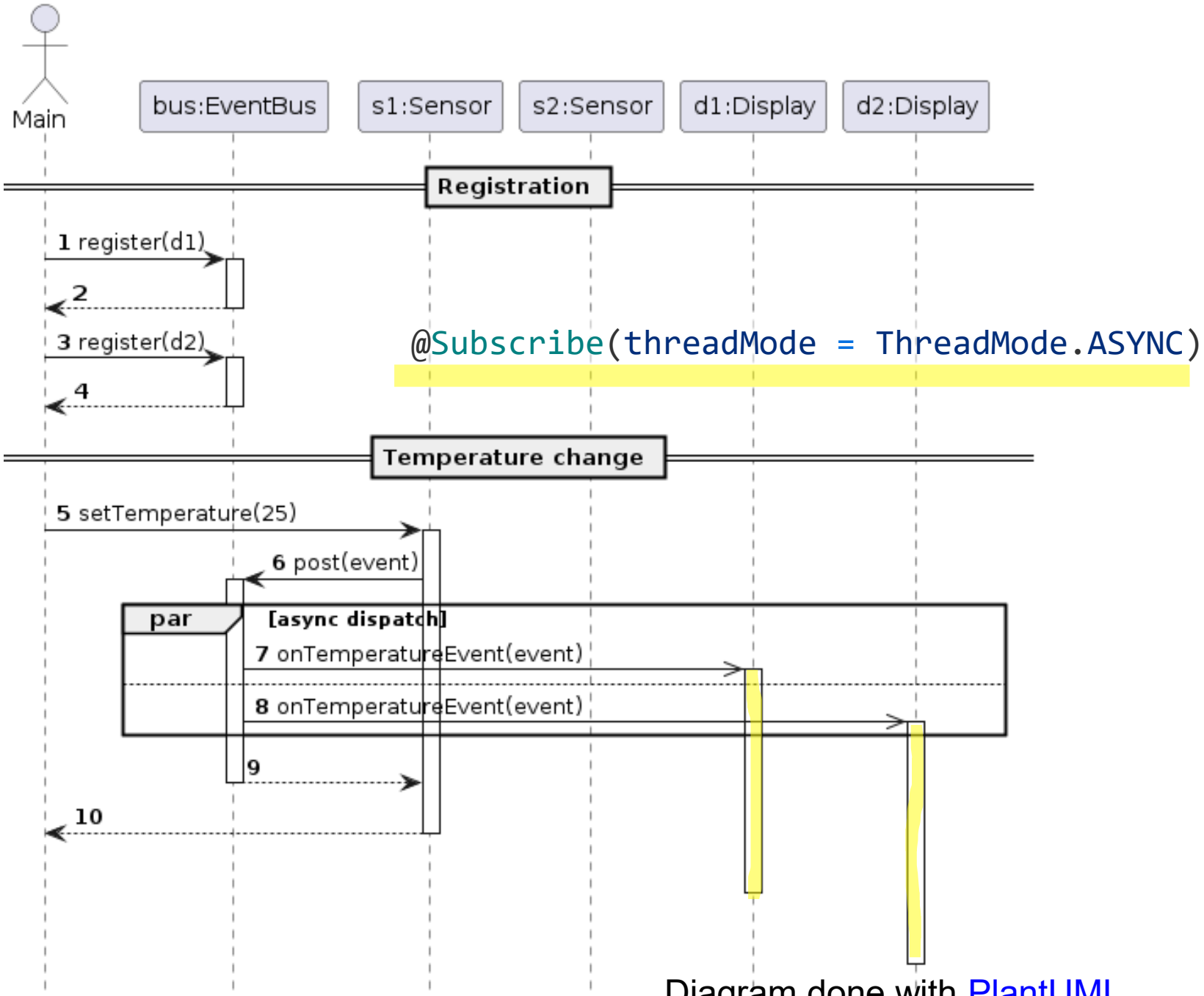


Diagram done with [PlantUML]

Thread handling in Greenrobot EventBus

- The execution thread depends on the ThreadMode used in @Subscribe.
- **Thread Modes:**
  - **POSTING (Default)**
    - Runs in the same thread that calls post(); No thread switching. (synchronous)
  - **MAIN**
    - Runs on the main/UI thread; Used mainly in Android
  - **BACKGROUND**
    - Runs in background thread if posted from main. Otherwise runs in posting thread
  - **ASYNC**
    - Always runs in a separate thread. Uses internal thread pool. Non-blocking for publisher




```
@Subscribe(threadMode = ThreadMode.ASYNC)
```

Diagram done with [PlantUML]

Async ThreadMode

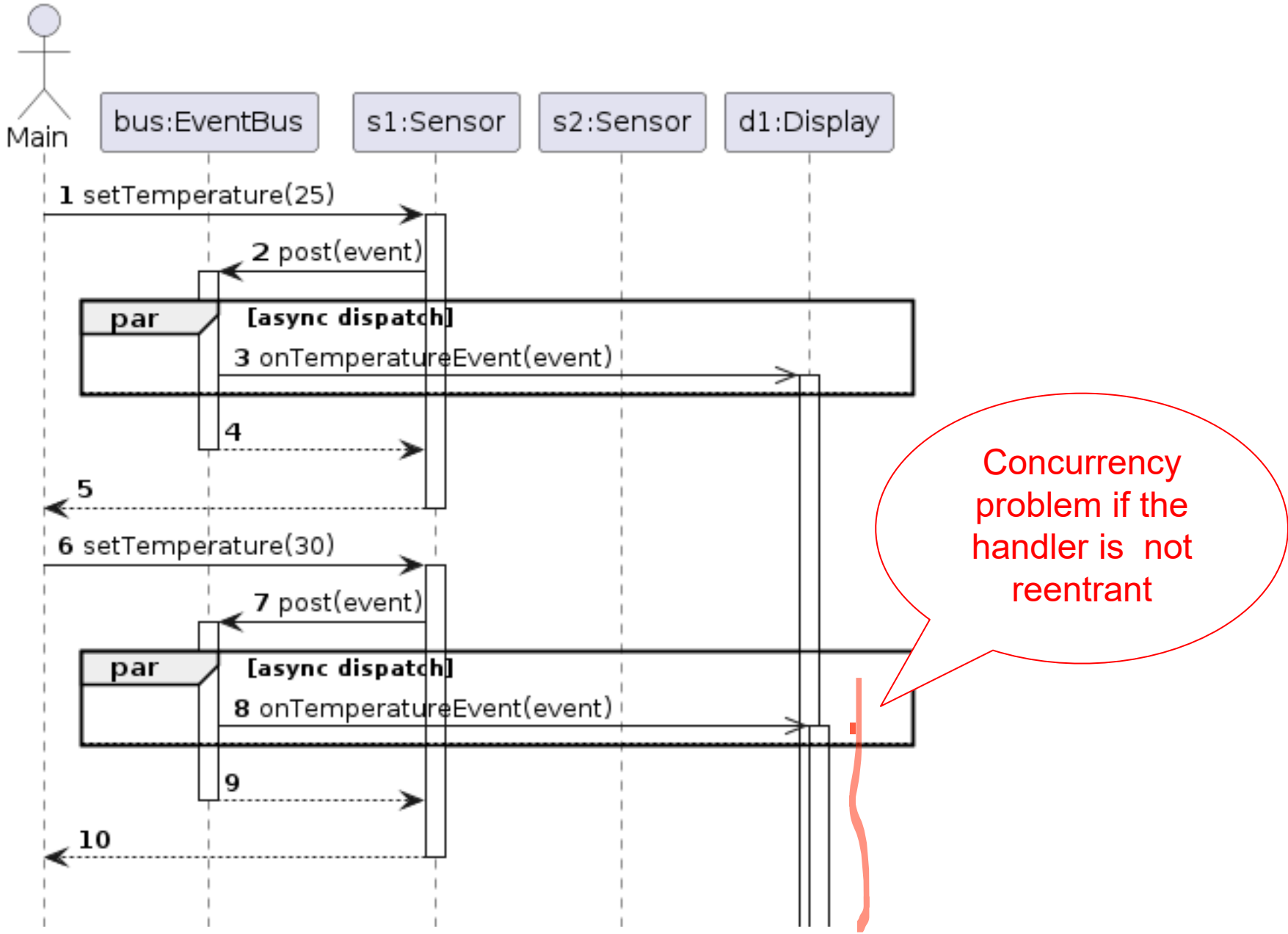
- Subscribers with ThreadMode = ASYNC will get their handler executed in a separate thread from a thread pool of the event bus
  - The posting thread (the sensor) does NOT wait until the subscriber(the display) finishes executing the handler
  - ASYNC mode is useful when the subscriber handler contains a time-consuming activity

Dangers in Async ThreadMode

- Caveat: the handler will be executed concurrently with other handlers, and maybe concurrently with another instance of itself!
  - if the handler accesses **shared state**, it is the responsibility of the programmer to protect it

```
@Subscribe (threadMode = ThreadMode.ASYNC)
public void onTemperatureEvent(TemperatureEvent event) {
  savedTemp  = event.getValue());
}
```

  - Best practices:
    - Make handlers stateless if possible (no shared mutable state)
    - Use thread-safe collections
    - Mutual exclusion (synchronized handler methods) – this removes much of the benefit of Async execution


Main
bus:EventBus
s1:Sensor
s2:Sensor
d1:Display
1 setTemperature(25)
2 post(event)
par
[async dispatch]
3 onTemperatureEvent(event)
4
5
6 setTemperature(30)
7 post(event)
par
[async dispatch]
8 onTemperatureEvent(event)
9
10
Concurrency problem if the handler is not reentrant

Example2: Multithreaded sensors

- Each sensor (publisher) runs in its thread and periodically posts new values
- EventBus accepts that multiple threads call post() simultaneously (the post()method is not synchronized)
- Subscribers (displays):
  - Subscribers with ThreadMode = POSTING will get their handler executed inside the sensor thread
    - If publishers run concurrently, then in POSTING mode the handlers may run concurrently => Handler code must be thread-safe (either reentrant or mutual exclusion)
  - Subscribers with ThreadMode = ASYNC will get their handler executed in a separate thread from a thread pool of the event bus
    - The same concurrency problems for handlers

```java
public class TemperatureSensor implements Runnable {

    private final EventBus eventBus;
    private final String id;
    private final String location;
    private final EnvironmentType environmentType;
    private final Random random = new Random();

    private volatile boolean running = true;

    public TemperatureSensor( … ) {
    //  … constructor details omitted
    }

    public void run() {
        while (running) {
            try {
                Thread.sleep(1000); // every 1000 ms
                float temperature = -10 + random.nextFloat() * 50;
                TemperatureEvent event =  new TemperatureEvent(id,
                        location, temperature, environmentType);
                eventBus.post(event);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                running = false;
            }
        }
```

```
public class MainSensors {

    public static void main(String[] args) throws InterruptedException  {

        EventBus eventBus = EventBus.getDefault();

        GeneralDisplay d1 =  new GeneralDisplay("Display1",
                Set.of(EnvironmentType.CITY, EnvironmentType.FOREST));
        eventBus.register(d1);

         // similar  create and register displays d2, d3,  d4

        TemperatureSensor ts1 =  new TemperatureSensor(eventBus, "TS001",
                        "Arad", EnvironmentType.CITY);
        Thread t1 = new Thread(ts1);
        t1.start();

        // similar for sensors ts2, ts3, ts4 start threads t2 t3 t4

        Thread.sleep(10000);  // let run for a while

        ts1.stop();  ts2.stop();  ts3.stop();   ts4.stop();

        t1.join();  t2.join();  t3.join();   t4.join();

    }
}
```

Source Code Download

- src.zip
- Greenrobot maven dependency:

```
<dependency>
    <groupId>org.greenrobot</groupId>
    <artifactId>eventbus-java</artifactId>
    <version>3.3.1</version>
</dependency>
```