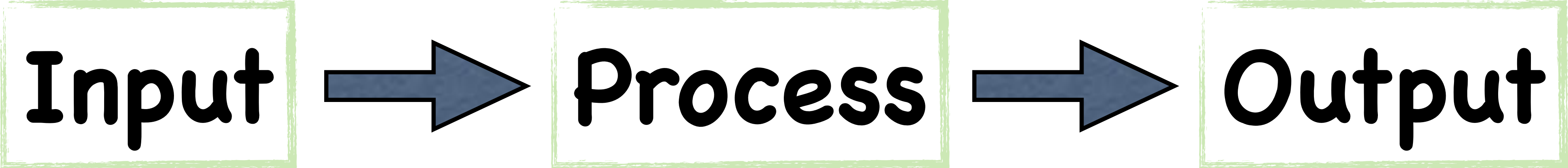
Introduction to Event-Driven Programming

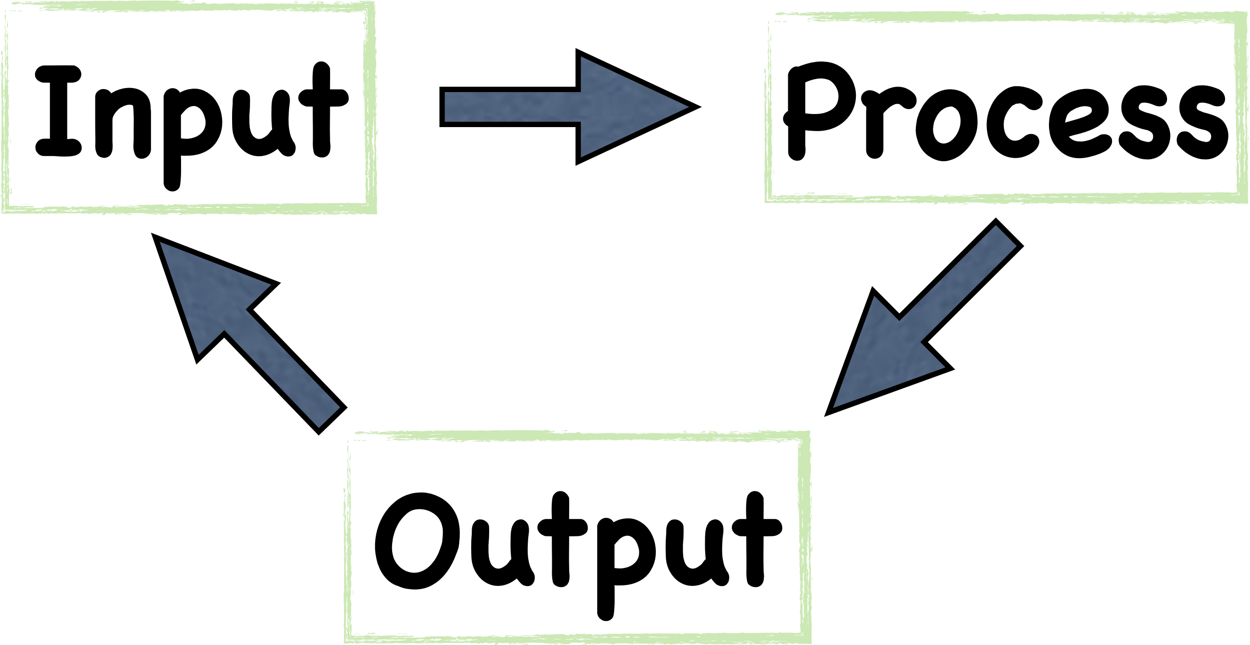
Samir Genaim

User-Program Interaction

- A program in its simplest form is a sequence of instructions, executed from start to end



- A more elaborated view is one in which the control flow is driven by the user's input



```
while ( … ) {
    String x = read();
    if (x="…")
        f1(…)
    else if (x=="…")
        f2(….)
    else ….
}
```

The Sensors Example

Write a program that reads the temperature of a room from a **temperature sensor**, if the temperature **falls below 20** it should turn the **Heater** on and the **Air Conditioner** off, and the other way around if it **goes above 30**.

Solution I - part I

```java
public class AirCondition {

    private boolean isOn;
    String id;

    public AirCondition(String id) {
        isOn = false;
        this.id = id;
    }

    public void on() {
        isOn = true;
        System.out.println("AC " + id + ": on");
    }

    public void off() {
        isOn = false;
        System.out.println("AC " + id + ": off");
    }
}
```

see: examples.sensors.ver0

Solution I - part II

```
public class Heater {

    private boolean isOn;
    String id;
    public Heater(String id) {
        isOn = false;
        this.id = id;
    }
    public void on() {
        isOn = true;
        System.out.println("HT " + id + ": on");
    }
    public void off() {
        isOn = false;
        System.out.println("HT" + id + ": off");
    }
}
```

see: examples.sensors.ver0

Solution I - part III

```
public class TemperatureSensor {

   private float t;
   private boolean running;
   private String id;

   public TemperatureSensor(String id) {
      running = true;
      this.id = id;
   }

   public float getTemperature() {
      t = HWLib.getTemperature(id);
      return t;
   }

}
```

details of HWLib.getTemperature are not important for now, assume it reads it from a file to which an actual sensor writes the required temperature, etc.

see: examples.sensors.ver0

Solution I - part IV

```
public class Main {
    public static void main(String[] args) {
        TemperatureSensor s = new TemperatureSensor("s");
        Heater h = new Heater("h");
        AirCondition a = new AirCondition("a");

        while ( true ) {
            float currTemp = s.getTemperature();
            if ( currTemp > 30 ) {
                h.off(); a.on();
            } else if ( currTemp < 20 ) {
                h.on(); a.off();
            }
            sleep(5000); // wait 5 seconds
        }
    }
}
```

see: examples.sensors.ver0

Solution I - drawbacks

```
public class Main {
    public static void main(String[] args) {
        TemperatureSensor s = new TemperatureSensor("s");
        Heater h = new Heater("h");
        AirCondition a = new AirCondition("a");

        while ( true ) {
            float currTemp = s.getTemperature();
            if ( currTemp > 30 ) {
                h.off();    a.on();
            } else if ( currTemp < 20 ) {
                h.on(); a.off();
            }
            sleep(5000); // wait 5 seconds
        }
    }
}
```

Adding a Heater or an AirConidion ...

```
Heater h1 = new Heater("h");
AirCondition a1 = new AirCondition("a");
```

```
h1.on(); a1.off();
```

```
h1.off(); a1.on();
```

... requires "deep" modifications of the code, which violates the open/closed principle of OOP

open/closed principle

software entities (classes, methods, etc.) should be open for extension, but closed for modification

see: examples.sensors.ver0

Solution II - part I

```
public class Main {
   public static void main(String[] args) {
      TemperatureSensor s = new TemperatureSensor("s");
      Heater h = new Heater("h",19,30);
      AirCondition a = new AirCondition("a",30,19);

      s.registerTempObserver(a);
      s.registerTempObserver(h);

      while ( true ) {
         s.refresh();
         sleep(5000);
      }
   }
}
```

a and **h** register in **s** to be **notified** when the temp. change -- and they react when they are notifed

s notify all registered objects when an event occur (event is temperature change)

see: examples.sensors.ver1

Solution II - Part II

```
s.registerTempObserver(a);
s.registerTempObserver(h);
```

- The method **registerTempObserver** (of the sensor class) must be able to receive an object of type Heater or AirCondition, or any other device that is interested in being notified.
- IMPORTANT: we **don't want** to define such a method for each device: **registerACTempObs**, **registerHeaterTempObs**, etc.
- Abstraction is the solution!

see: examples.sensors.ver1

Solution II - Part III

Define an interface to be implemented by the device classes, and **registerTempObserver** will use this interface for its parameter.

```
public interface TempObserver {
   public void tempChanged(float t);
}
```

Typically we do a similar abstraction for the sensors, to declare that "I am a class that can provide you with the temperature" -- will become clear later why this is useful.

```
public interface TempObservable {
   public void registerTempObserver(TempObserver t);
}
```

see: examples.sensors.ver1

Solution II - Part IV

```
public class AirCondition implements TempObserver {
   private boolean isOn;
   private float onTemp;
   private floar offTemp;
   private String id;

   public AirCondition(String id, float onTemp, float offTemp) {
      isOn = false;
      this.id = id;
      this.onTemp = onTemp;
      this.offTemp = offTemp;
   }

   ...

   @Override
   public void tempChanged(float t) {
      if (t < this.offTemp && isOn) off();
      else if (t > onTemp && !isOn) on();
   }
}
```

When notified about temp. change, it does something

see: examples.sensors.ver1

Solution II - Part V

```
public class Heater implements TempObserver {

    private boolean isOn;
    private float onTemp;
    private floar offTemp;
    private String id;

    public Heater(String id, float onTemp, float offTemp) {
        isOn = false;
        this.id = id;
        this.onTemp = onTemp;
        this.offTemp = offTemp;
    }

    ...

    @Override
    public void tempChanged(float t) {
        if (t > this.offTemp && isOn) off();
        else if (t < onTemp && !isOn) on();
    }
}
```

When notified about temp. change, it does something

see: examples.sensors.ver1

Solution II - Part VI

```java
public class TempSensor implements TempObservable {
   private float t;
   private List<TempObserver> obs;
   private String id;
   public TemperatureSensor(String id) {
      obs = new ArrayList<TempObserver>();
      this.id = id;
      refresh();
   }
   public void refresh() {
      float x = HWLib.getTemperature(id);
      if (t != x) {
         t = x;
         for (TempObserver o : obs)  o.tempChanged(t);
      }
   }
   public void registerTempObserver(TempObserver l) {
      obs.add(l);
   }
}
```

A list of observers

When the temperature changes, it notifies all observers

When an observer registers, it is added to the list of observers

see: examples.sensors.ver1

Observe Several Events

Observes can have several methods or encapsulate the type of notification in an Event class

```
public interface TempObserver {
   public void tempChanged(float t);
   public void tempLow(float t);
   public void tempHigh(float t);
}
```

```
public interface TempObserver {
   public void notify(Event t);
}
```

Observables might also allow registering for different kinds of notifications

```
public interface TempObservable {
   public void registerTempObserver(TempObserver t);
   public void registerHighTempObserver(TempObserver t);
   public void registerLowTempObserver(TempObserver t);
}
```

see: examples.sensors.ver1

Important Design Principles

- Don’t produce unnecessary events (fewer calls are better), calls might be expensive!!
- Handlers should react quickly, otherwise the application become non-responsive.
- In our example we generate an event whenever the temperature changes -- too many!!. Moreover, the listeners react only for some specific values.
- We could redesign the application to send notification only when the temperature reaches some limit.

Event-Based Programming and GUI

- Modern Graphical User Interfaces libraries, heavily rely on even-based programming.

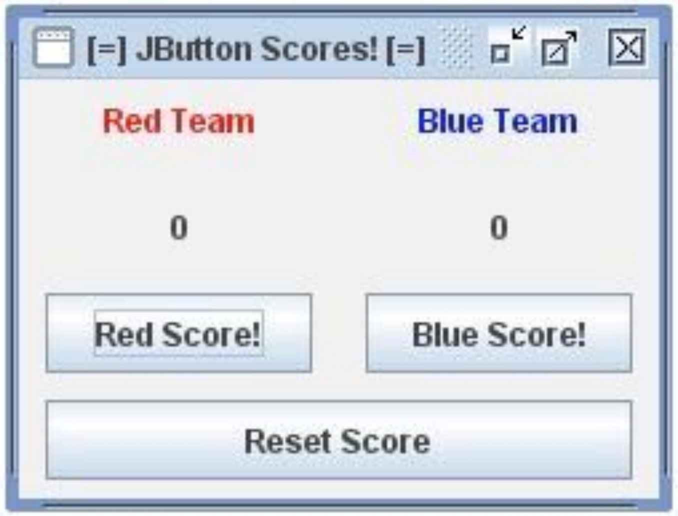
  Events are the actions that the user does on the elements of the GUI:

  1. Button pressed
  2. Select an item from a menu
  3. Mouse click somewhere on a given window
  4. etc.

- Buttons, Menus, etc., are similar to the Sensor
- The user handlers are similar to the Heater, AC, etc.

GUI (swing) Example

```
class ButtonExample implements ActionListener {
    public JPanel createContentPane() {
        blueButton = new JButton("Blue Score!");
        blueButton.addActionListener(this);
        ...
    }
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == blueButton) {
            // do something
        } else if ...
    }
}
```

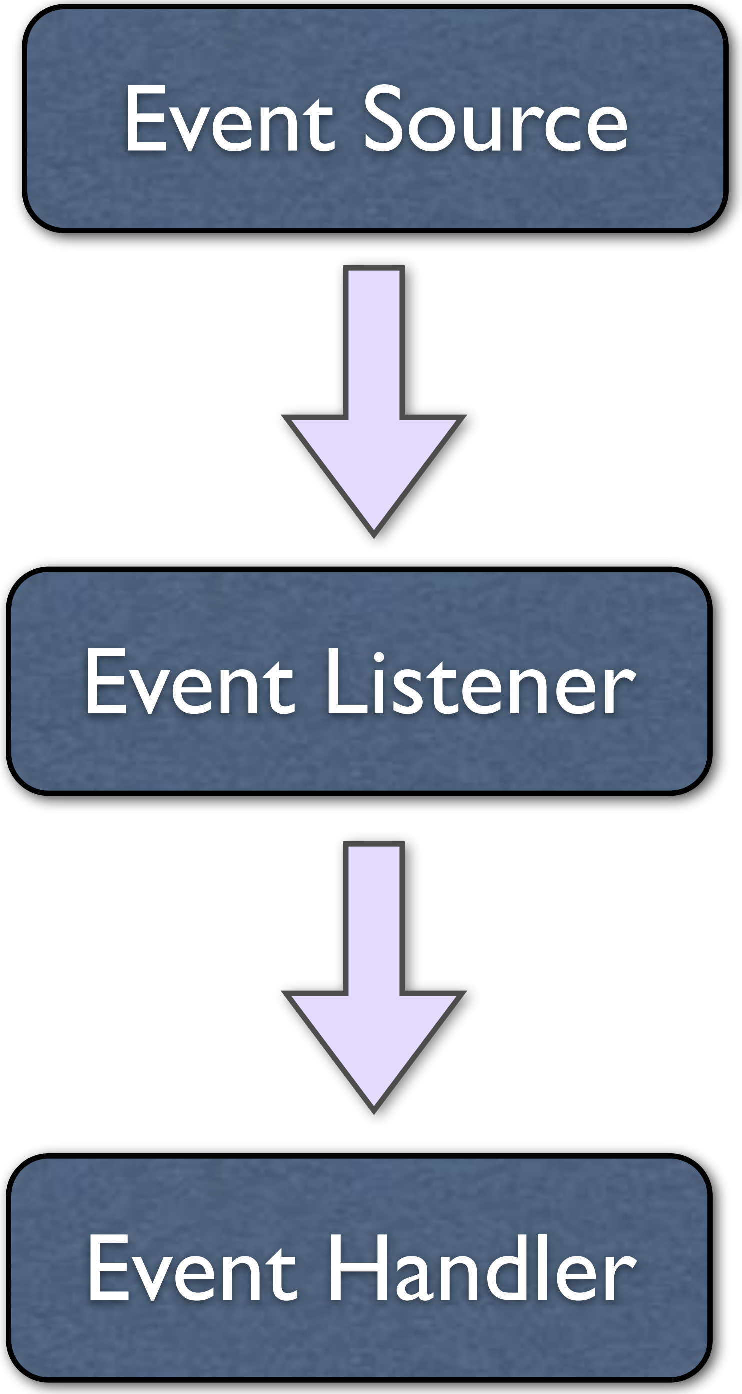


When blueButton is clicked, it calls back method actionPerformed on object this.

Events/Listeners are Very General

- Anything can be an event, and any response can be programmed as listener.
- Object A register itself in object B to be called when something happens
  1. a field has been updated
  2. a field has been assigned some specific values
  3. some error occurred, etc.
- Event-based programming simplifies the way messages are passed between objects.
- Non-centralised treatment of control-flow. Each object is responsible for its own events only. We don't have a global loop that controls what to execute next.

Summary



1. Listeners must register to receive notifications when events occur
2. The event source "calls back" the listeners when the events happen, possibly adding some information on the event
3. The listeners decide how to act, and call some method to handle the event
4. In Java all this mechanism is done with interfaces
5. Aim at few messages, and quick handlers

What we have seen in this lecture is actually a well-known design pattern:

The Observer Design Pattern

We will come back to this important design pattern later ...