



Event Driven Programming in Java

Event-driven programming

- ▶ The flow of the program is determined by events
- ▶ It is the dominant paradigm used in graphical user interfaces and web applications
- ▶ Centered on performing certain actions in response to user input
- ▶ Events such as:
 - ▶ user actions
 - ▶ mouse clicks
 - ▶ key presses
 - ▶ sensor outputs
 - ▶ messages from other programs/threads

Event driven programming

- Program waits for events
- Whenever something happens the program responds and does something



How to “pass” functionality in Java?

- ▶ Using ordinary objects
- ▶ Using anonymous objects
- ▶ Using lambda expressions (Java 8)
- ▶ Using method references (Java 8)
- ▶ Using reflection (lets not do this, yet...)

Using ordinary objects

```
class PassFun1 {
    // Method that takes a "method" as argument
    static void exampleMethod(Runnable toRun) {
        toRun.run();
    }
    public static void main(String[] args) {
        MyObject obj1 = new MyObject();
        exampleMethod(obj1);
    }
}

class MyObject implements Runnable{
    @Override
    public void run() {
        System.out.println("Hello students!");
    }
}
```

Command Prompt

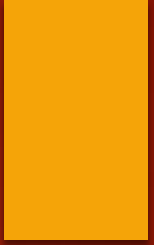
```
E:\myjavaprogs>javac PassFun1.java
```

```
E:\myjavaprogs>java PassFun1  
Hello students!
```

```
E:\myjavaprogs>_
```

Using anonymous objects

```
class PassFun2 {  
    // Method that takes a "method" as argument  
    static void exampleMethod(Runnable toRun) {  
        toRun.run();  
    }  
    public static void main(String[] args) {  
        exampleMethod(new Runnable() {  
            @Override  
            public void run() {  
                System.out.println("Hello Students 2!");  
            }  
        });  
    }  
}
```



```
Command Prompt
E:\myjavaprogs>javac PassFun2.java
E:\myjavaprogs>java PassFun2
Hello Students 2!
E:\myjavaprogs>_
```


Using lambda expressions (Java 8)

```
class PassFun3 {  
    // Method that takes a "method" as argument  
    static void exampleMethod(Runnable toRun) {  
        toRun.run();  
    }  
    public static void main(String[] args) {  
        exampleMethod(() -> System.out.println("Hello Students 3!"));  
    }  
}
```

Command Prompt

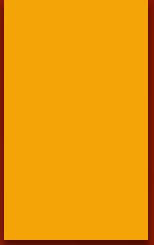
```
E:\myjavaprogs>javac PassFun3.java
```

```
E:\myjavaprogs>java PassFun3  
Hello Students 3!
```

```
E:\myjavaprogs>_
```

Using method references (Java 8)

```
class PassFun4 {  
    // Method that takes a "method" as argument  
    static void exampleMethod(Runnable toRun) {  
        toRun.run();  
    }  
    static void helloStudents() {  
        System.out.println("Hello Students 4!");  
    }  
    public static void main(String[] args) {  
        exampleMethod(PassFun4::helloStudents);  
    }  
}
```



```
Command Prompt
E:\myjavaprogs>javac PassFun4.java
E:\myjavaprogs>java PassFun4
Hello Students 4!
E:\myjavaprogs>_
```

Example

```
import java.util.*;
class PassFunExample {
    public static void main(String[] args) {
        Map<Character, Runnable> commands = new HashMap<>();
        // Populate commands map
        commands.put('h', () -> System.out.println("Type h or q"));
        commands.put('q', () -> System.exit(0));
        while (true) {
            // Print menu
            System.out.println("Main Menu");
            System.out.println("Please Choose!");
            System.out.println("h) Help");
            System.out.println("q) Quit");
            // User input
            char key = new Scanner(System.in).nextLine().charAt(0);
            // Run selected command
            if (commands.containsKey(key))
                commands.get(key).run();
        }
    }
}
```

```
Command Prompt
E:\myjavaprogs>javac PassFunExample.java
E:\myjavaprogs>java PassFunExample
Main Menu
Please Choose!
h) Help
q) Quit
h
Type h or q
Main Menu
Please Choose!
h) Help
q) Quit
q
E:\myjavaprogs>_
```

A simple Event Listener Interface

```
public interface EventListener {  
    public void onSomeChange(State oldState, State newState);  
}
```

A simple class using the listener Interface

```
public class EventOwner {  
    public void addEventListener(EventListener listener) { ... }  
}
```


Implementation in Java 7

Anonymous Interface Implementation!

```
EventOwner eventOwner = new EventOwner();
eventOwner.addEventListener(new EventListener() {
    public void onSomeChange(State oldState, State newState) {
        // do something with the old and new state.
    }
});
```

Implementation in Java 8

Java Lambda
Expression!

```
EventOwner eventOwner = new EventOwner();  
  
eventOwner.addEventListener(  
    (oldState, newState) -> System.out.println("Something changed!")  
);
```

Lambda expression usage

- ▶ The lambda expression is matched against the parameter type of the `addEventListener()` method's parameter
- ▶ If the lambda expression matches the parameter type (in this case the `EventListener` interface) , then the lambda expression is turned into a function that implements the same interface as that parameter.

Matching Lambdas and Interfaces

- ▶ A single method interface is also sometimes referred to as a functional interface
- ▶ We have to follow 3 rules
 - ▶ The interface should have only one method
 - ▶ The parameters of the lambda expression should match the parameters of the single method
 - ▶ The return type of the lambda expression should match the return type of the single method

Lambda Expressions with Zero Parameters

```
() -> System.out.println("Zero parameter lambda");
```

Lambda Expressions with One Parameter

```
(param) -> System.out.println("One parameter: " + param);
```

or

```
param -> System.out.println("One parameter: " + param);
```

Lambda Expressions with Multiple Parameters

```
(p1, p2) -> System.out.println("Multiple parameters: " + p1 + ", " + p2);
```

Lambda Expression Parameter Types

- ▶ Specifying parameter types for a lambda expression may be necessary if the compiler cannot infer the parameter types from the functional interface method the lambda expression is matching

```
(Student student1) -> System.out.println("Student's name is: " + student1.getName());
```


Lambda Expression Function Body One Line

```
(oldState, newState) -> System.out.println("Something changed!")
```

Lambda Expression Function Body Multiple Lines

```
(oldState, newState) -> {  
    System.out.println("Old state: " + oldState);  
    System.out.println("New state: " + newState);  
}
```

Lambda Expression Returning Value

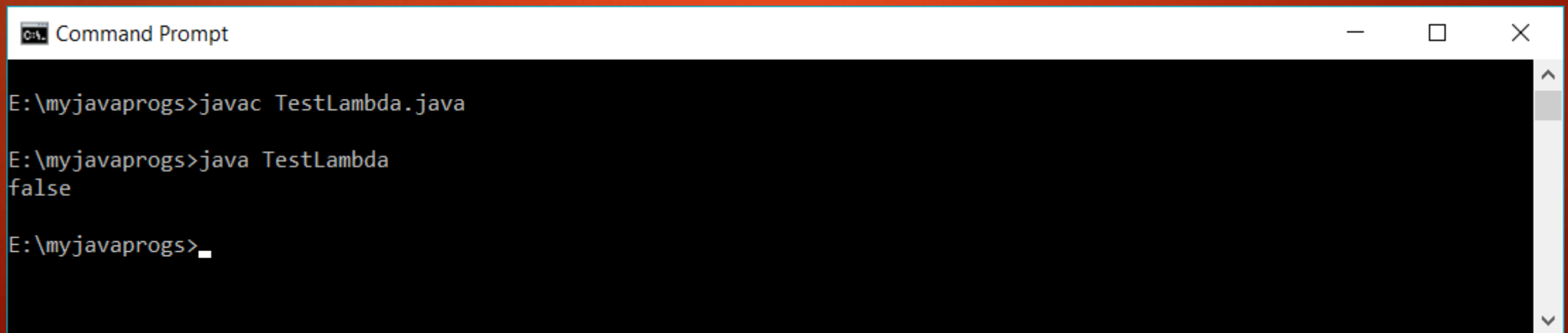
```
(param) -> {  
    System.out.println("param: " + param);  
    return "some value";  
}
```

Lambda Expressions as Objects

```
public interface MyComparator {  
    public boolean compare(int a1, int a2);  
}  
  
MyComparator myComparator = (a1,a2) -> {return a1 > a2;};  
  
boolean result = myComparator.compare(5, 10);
```

Demo

```
class TestLambda {  
    public static void main(String[] args) {  
        MyComparator myComparator = (a1,a2) -> a1>a2;  
        boolean result = myComparator.compare(5, 10);  
        System.out.println(result);  
    }  
}  
  
interface MyComparator {  
    public boolean compare(int a1, int a2);  
}
```



```
Command Prompt  
E:\myjavaprogs>javac TestLambda.java  
  
E:\myjavaprogs>java TestLambda  
false  
  
E:\myjavaprogs>_
```

Creating a custom Event and Event Listener

Main Components

- ▶ An interface to be implemented by everyone interested in the custom events
- ▶ A class that fires these specific custom events
- ▶ A class that is interested in listening for the custom events
- ▶ And...a test class

A simple Interface first

```
interface HelloListener {  
    void someoneSaidHello();  
}
```


A class that fires events

```
class Initiater {  
    private List<HelloListener> listeners = new ArrayList<HelloListener>();  
    public void addListener(HelloListener toAdd) {  
        listeners.add(toAdd);  
    }  
    public void sayHello() {  
        System.out.println("Hello! Anyone there?");  
        // Notify everybody that may be interested.  
        for (HelloListener hl : listeners)  
            hl.someoneSaidHello();  
    }  
}
```

A class (or more) that are interested in listening to the events

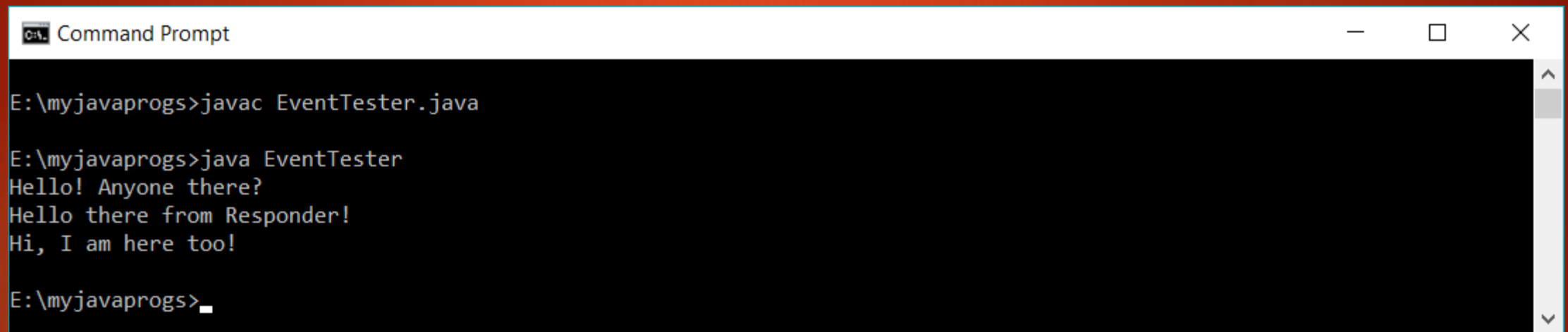
```
// Someone interested in "Hello" events
class Responder implements HelloListener {
    @Override
    public void someoneSaidHello() {
        System.out.println("Hello there from Responder!");
    }
}

// Someone else too
class AnotherResponder implements HelloListener {
    @Override
    public void someoneSaidHello() {
        System.out.println("Hi, I am here too!");
    }
}
```

A demo use case

```
class EventTester {  
    public static void main(String[] args) {  
        Initiater initiator = new Initiater();  
        Responder responder = new Responder();  
        AnotherResponder responder2 = new AnotherResponder();  
        initiator.addListener(responder);  
        initiator.addListener(responder2);  
        initiator.sayHello();  
    }  
}
```

Run the example!



```
Command Prompt
E:\myjavaprogs>javac EventTester.java
E:\myjavaprogs>java EventTester
Hello! Anyone there?
Hello there from Responder!
Hi, I am here too!
E:\myjavaprogs>_
```

Time to see a more “complete”
example

Command Prompt

```
E:\myjavaprogs\CustomEventJava>javac CarDemo.java
```

```
E:\myjavaprogs\CustomEventJava>java CarDemo
```

```
Alert! You have exceeded 40 MPH!
```

```
Alert! You have exceeded 90 MPH!
```

```
Uhm... you are driving 10 MPH. Speed up!
```

```
E:\myjavaprogs\CustomEventJava>
```



Lets do the same using
Java Built-in classes

OBSERVER - OBSERVABLE

Class Observable

- ▶ This class represents an observable object, or "data" in the model-view paradigm.
- ▶ It can be subclassed to represent an object that the application wants to have observed.
- ▶ An observable object can have one or more observers.
- ▶ An observer may be any object that implements interface Observer.
- ▶ After an observable instance changes, an application calling the Observable's notifyObservers method causes all of its observers to be notified of the change by a call to their update method.

Observable Method Summary

| Modifier and Type | Method and Description |
|-------------------|---|
| void | addObserver(Observer o) Adds an observer to the set of observers for this object, provided that it is not the same as some observer already in the set. |
| protected void | clearChanged() Indicates that this object has no longer changed, or that it has already notified all of its observers of its most recent change, so that the <code>hasChanged</code> method will now return <code>false</code> . |
| int | countObservers() Returns the number of observers of this <code>Observable</code> object. |
| void | deleteObserver(Observer o) Deletes an observer from the set of observers of this object. |
| void | deleteObservers() Clears the observer list so that this object no longer has any observers. |
| boolean | hasChanged() Tests if this object has changed. |
| void | notifyObservers() If this object has changed, as indicated by the <code>hasChanged</code> method, then notify all of its observers and then call the <code>clearChanged</code> method to indicate that this object has no longer changed. |
| void | notifyObservers(Object arg) If this object has changed, as indicated by the <code>hasChanged</code> method, then notify all of its observers and then call the <code>clearChanged</code> method to indicate that this object has no longer changed. |
| protected void | setChanged() Marks this <code>Observable</code> object as having been changed; the <code>hasChanged</code> method will now return <code>true</code> . |

Interface Observer

- ▶ A class can implement the Observer interface when it wants to be informed of changes in observable objects

update

```
void update(Observable o,  
            Object arg)
```

This method is called whenever the observed object is changed. An application calls an `Observable` object's `notifyObservers` method to have all the object's observers notified of the change.

Parameters:

`o` - the observable object.

`arg` - an argument passed to the `notifyObservers` method.

Demo time!

An observable object

```
import java.util.Observable;

public class ObservableObject extends Observable
{
    private String weather;

    public ObservableObject(String weather)
    {
        this.weather = weather;
    }
    public String getWeather()
    {
        return weather;
    }
    public void setWeather(String weather)
    {
        this.weather = weather;
        setChanged();
        notifyObservers();
    }
}
```

An observer object

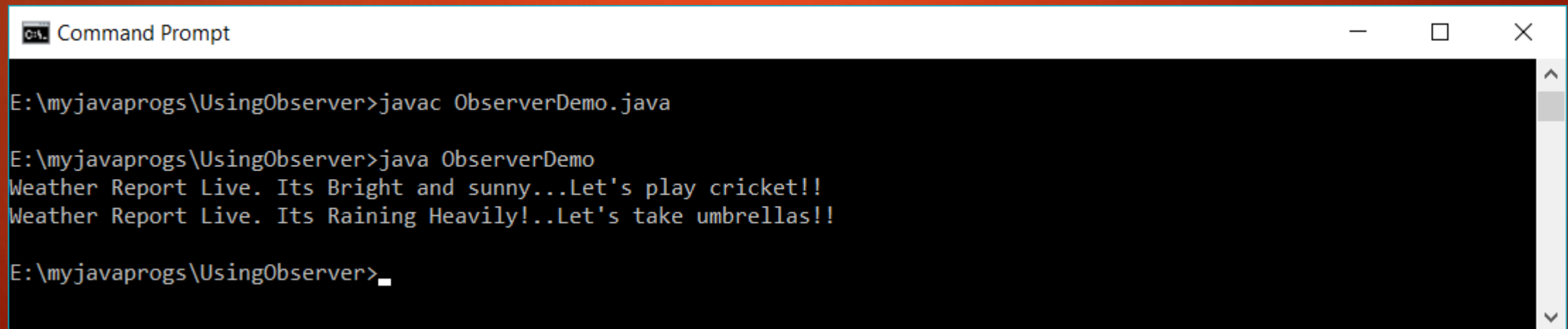
```
import java.util.Observable;
import java.util.Observer;

class ObserverObject implements Observer
{
    private ObservableObject weatherUpdate ;

    @Override
    public void update(Observable observable, Object arg)
    {
        weatherUpdate = (ObservableObject) observable;
        System.out.println("Weather Report Live. Its "+weatherUpdate.getWeather());
    }
}
```

Test it!

```
public class ObserverDemo{  
  
    public static void main(String[] args)  
    {  
        ObservableObject observable = new ObservableObject("Cloudy");  
        ObserverObject observer = new ObserverObject();  
        observable.addObserver(observer);  
        observable.setWeather("Bright and sunny...Let's play cricket!! ");  
        observable.setWeather("Raining Heavily!..Let's take umbrellas!!");  
    }  
}
```



```
Command Prompt  
E:\myjavaprogs\UsingObserver>javac ObserverDemo.java  
E:\myjavaprogs\UsingObserver>java ObserverDemo  
Weather Report Live. Its Bright and sunny...Let's play cricket!!  
Weather Report Live. Its Raining Heavily!..Let's take umbrellas!!  
E:\myjavaprogs\UsingObserver>_
```