



Università degli Studi dell'Insubria  
Dipartimento di Scienze Teoriche e Applicate

---

# Programmazione Concorrente e Distribuita Gestione degli eventi

Luigi Lavazza  
Dipartimento di Scienze Teoriche e Applicate  
[luigi.lavazza@uninsubria.it](mailto:luigi.lavazza@uninsubria.it)

---



## Peer-to-peer

---

- RMI mette a disposizione la modalità peer-to-peer che permette l'invocazione reciproca tra due oggetti remoti richiedendo che solo uno di essi (il server) si registri con il **rmiregistry**.



# Funzionamento del peer-to-peer con due nodi

---

- Un client effettua il collegamento con il server remoto nel solito modo
  - ▶ Cioè facendo **lookup** sul **registry**
- Il client stesso diventa un oggetto remoto (quindi potenzialmente invocabile dal server)
  - ▶ mediante la chiamata  
`java.rmi.server.UnicastRemoteObject.exportObject(this, port);`
  - ▶ o automaticamente, se estende **UnicastRemoteObject**
- Il client passa il proprio riferimento al server, come parametro di un metodo remoto del server.



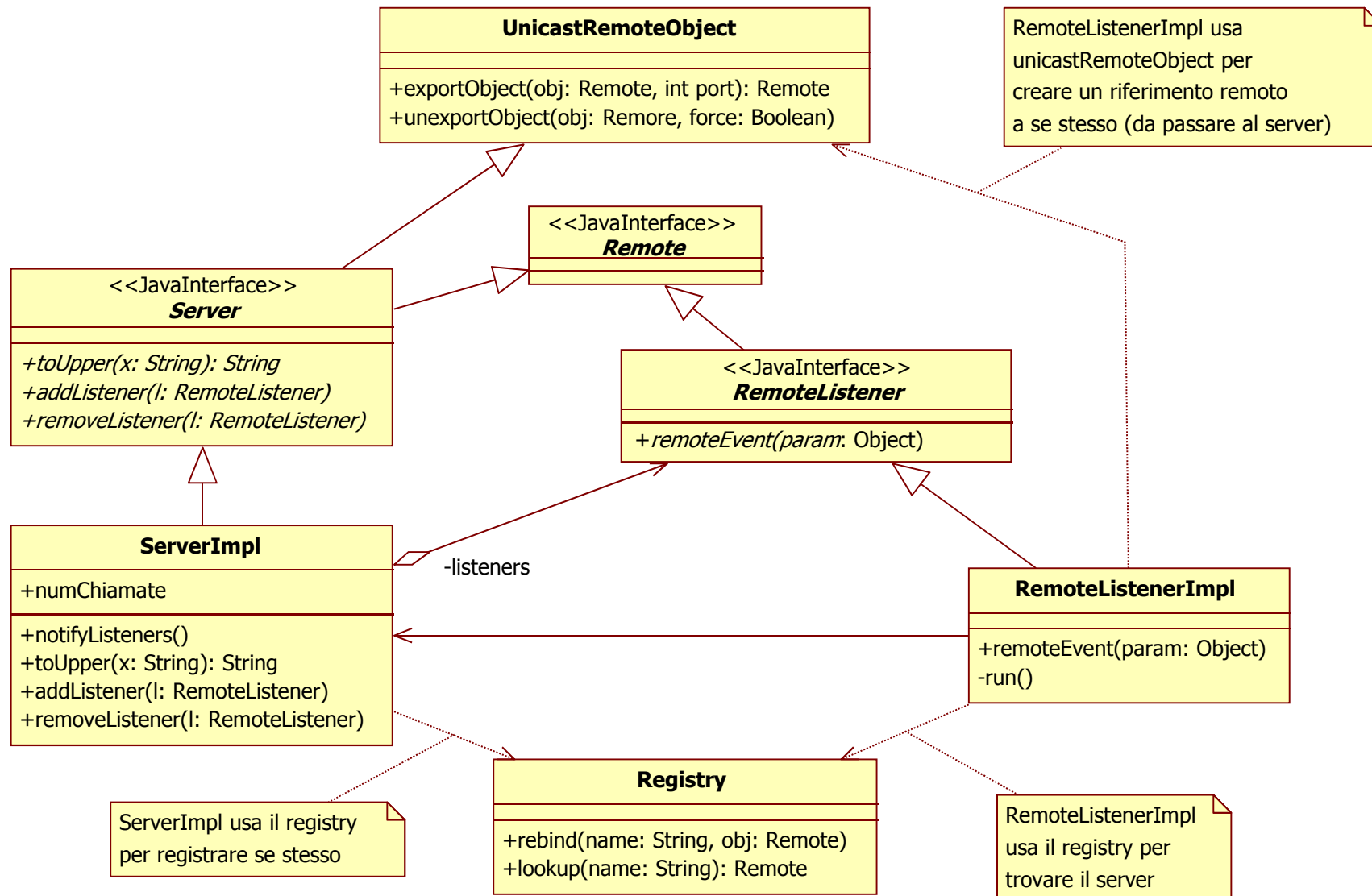
## Esempio

---

- L'esempio che segue è molto semplice e forse anche un po' stupido.
- Il servizio offerto dal server consiste nel convertire in maiuscolo i caratteri di una stringa data.
- In più il server comunica ai client quante volte che il servizio è stato chiamato

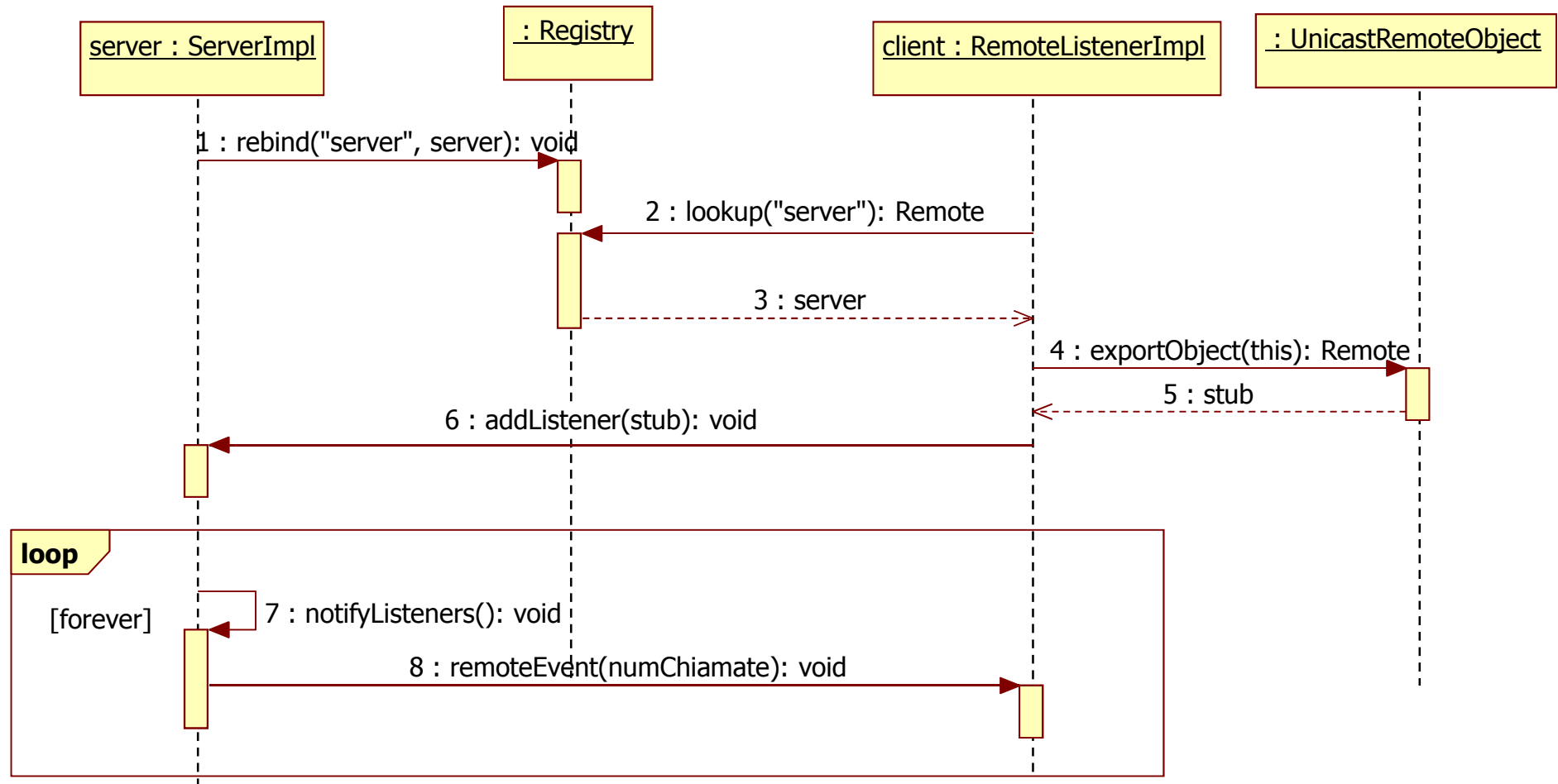


# Esempio





# Esempio





# Interfaccia Server

```
import java.rmi.Remote;  
import java.rmi.RemoteException;  
  
public interface Server extends Remote {  
    public String toUpperCase(String x)  
        throws RemoteException;  
    public void addListener(RemoteListener l)  
        throws RemoteException;  
    public void removeListener(RemoteListener l)  
        throws RemoteException;  
}
```

Servizi  
offerti

Metodi di  
gestione  
dei client



## Interfaccia RemoteListener (client)

---

```
import java.rmi.Remote;  
import java.rmi.RemoteException;  
  
interface RemoteListener extends Remote {  
    public void remoteEvent(Object param)  
        throws RemoteException;  
}
```

Metodo per  
ricevere notifiche





# Implementazione Server

---

```
import java.io.*;
import java.util.*;
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class ServerImpl extends UnicastRemoteObject
    implements Server {
    private List<RemoteListener> listeners =
        new ArrayList<RemoteListener>();
    private int numChiamate = 0;
    public ServerImpl() throws RemoteException {}
```



# Implementazione Server

---

```
public String toUpperCase(String x)
                                throws RemoteException{
    numChiamate++;
    return x.toUpperCase();
}
public void addListener(RemoteListener listener)
                        throws RemoteException {
    listeners.add(listener);
}
public void removeListener(RemoteListener listener)
                        throws RemoteException {
    listeners.remove(listener);
}
```



# Implementazione Server

---

```
private void notifyListeners() {  
    for (RemoteListener l : listeners) {  
        try {  
            l.remoteEvent(numChiamate);  
        } catch (RemoteException ee) {  
            listeners.remove(l);  
        }  
    }  
}
```



# Implementazione Server

---

```
public static void main(String[] args)
                                throws Exception {
    ServerImpl server = new ServerImpl();
    System.err.println("Registering...");
    Registry registry = LocateRegistry.createRegistry(1099);
    registry.rebind("server", server);
    System.err.println("Registered");
    while(true) {
        server.notifyListeners();
        Thread.sleep(500);
    }
}
}
```



## Implementazione RemoteListener (client)

---

```
import java.io.*;
import java.util.*;
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class RemoteListenerImpl implements RemoteListener{
    public void remoteEvent(Object param)
        throws RemoteException {
        System.err.println("REMOTE NOTIFICATION: num calls = "
            + param);
    }
}
```



## Implementazione RemoteListener (client)

---

```
private void work() throws Exception {
    String host = null; // localhost
    Registry registry = LocateRegistry.getRegistry(host);
    Server server = (Server) registry.lookup("server");
    RemoteListener stub = (RemoteListener)
        UnicastRemoteObject.exportObject(this, 3939);
    server.addListener(stub);
    for(int i=0; i<3; i++) {
        System.err.println("Chiamata remota: " +
                           server.toUpperCase("test "+i));
        Thread.sleep(1000);
    }
    server.removeListener(this);
    UnicastRemoteObject.unexportObject(this, false);
}


public static void main(String[] args) throws Exception{
    RemoteListenerImpl client = new RemoteListenerImpl();
    client.work();
}
}
```




# Deploy

---


## ▼ Client

 RemoteListener.java


 RemoteListenerImpl.java

 Server.java

## ▼ Server

 RemoteListener.java

 Server.java

 ServerImpl.java



# Esecuzione

```
gigi@gigi-HP-EliteBook-850-G6: ~/Documents/Didattica/Prog_CD/Code/2021/Lez14/Esempio_iniziale/Server$ rmiregistry &
[1] 12067
gigi@gigi-HP-EliteBook-850-G6:~/Documents/Didattica/Prog_CD/Code/2021/Lez14/Esempio_iniziale/Server$ java ServerImpl
Registering...
Registered

gigi@gigi-HP-EliteBook-850-G6:~/Documents/Didattica/Prog_CD/Code/2021/Lez14/Esempio_iniziale/Client$ java RemoteListenerImpl
Chiamata remota: TEST 0
REMOTE NOTIFICATION: num calls = 1
REMOTE NOTIFICATION: num calls = 1
Chiamata remota: TEST 1
REMOTE NOTIFICATION: num calls = 2
REMOTE NOTIFICATION: num calls = 2
Chiamata remota: TEST 2
REMOTE NOTIFICATION: num calls = 3
REMOTE NOTIFICATION: num calls = 3
gigi@gigi-HP-EliteBook-850-G6:~/Documents/Didattica/Prog_CD/Code/2021/Lez14/Esempio_iniziale/Client$
```





## Quando notificare

---

- Finora abbiamo visto solo il meccanismo di notifica dei client.
- Nell'esempio il server notifica secondo un proprio criterio.
- Esiste un caso molto importante in cui bisogna notificare: quando accade un **evento** specifico, di cui i client devono essere informati.



# Gestione degli Eventi nelle interfacce grafiche

---

- Tutte le volte che digitiamo un carattere o facciamo un click con il mouse generiamo un evento
- Possono esserci diversi oggetti interessati a essere informati dell'evento
  - ▶ per poter reagire opportunamente



# Gestione degli Eventi: oggetti coinvolti

---

- Funzionamento generale della gestione degli eventi in AWT (Abstract Window Toolkit):
  - ▶ Si crea un Listener

```
public class MyListener implements ActionListener {}
```
  - ▶ Si implementano i metodi dell'interfaccia **ActionListener**

```
public void actionPerformed(ActionEvent e) {  
    . . . // code that reacts to the action . . .  
}
```
  - ▶ Si registra il Listener

```
MyListener x = new MyListener();  
...  
someComponent.addActionListener(x);
```
  - ▶ Quando si verifica un evento in **someComponent**, **x** riceve una notifica, in modo che le azioni opportune siano eseguite.



## Gestione degli Eventi: esempio

---

```
class MyPanel extends JPanel implements ActionListener{  
    ...  
    public void actionPerformed(ActionEvent e) {  
        // la risposta al click sul pulsante va qui  
        ...  
    }  
MyPanel panel = new MyPanel();  
JButton button = new JButton("Open");  
button.addActionListener(panel);
```

**MyPanel** è un listener

Il metodo **actionPerformed** di **MyPanel** verrà eseguito in risposta ad eventi

Una istanza di **MyPanel** viene aggiunta ai listener di **button** (cioè reagirà agli eventi generati da **button**)



## Alcuni Listener

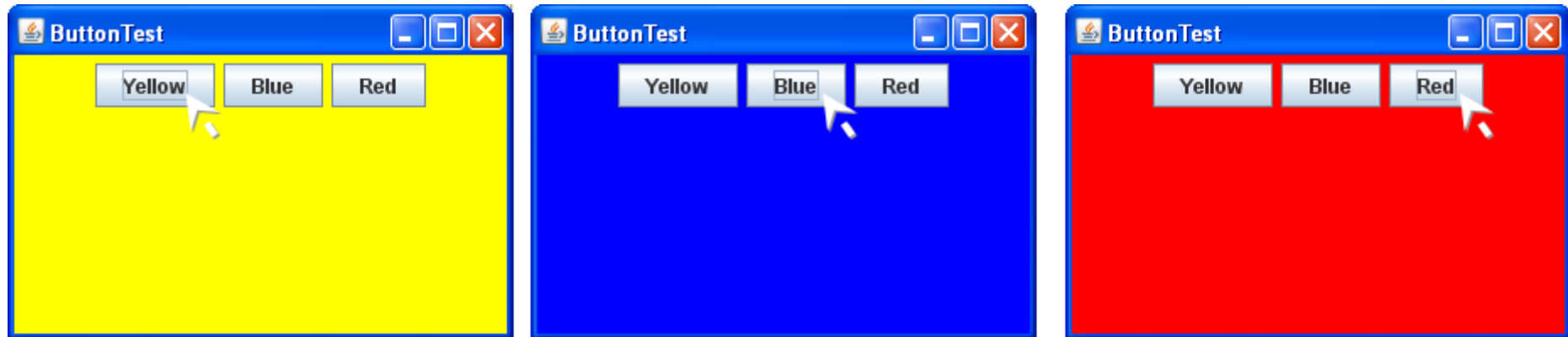
---

- Alcuni Listener legati ad azioni tipiche delle applicazioni Java con Interfaccia Grafica

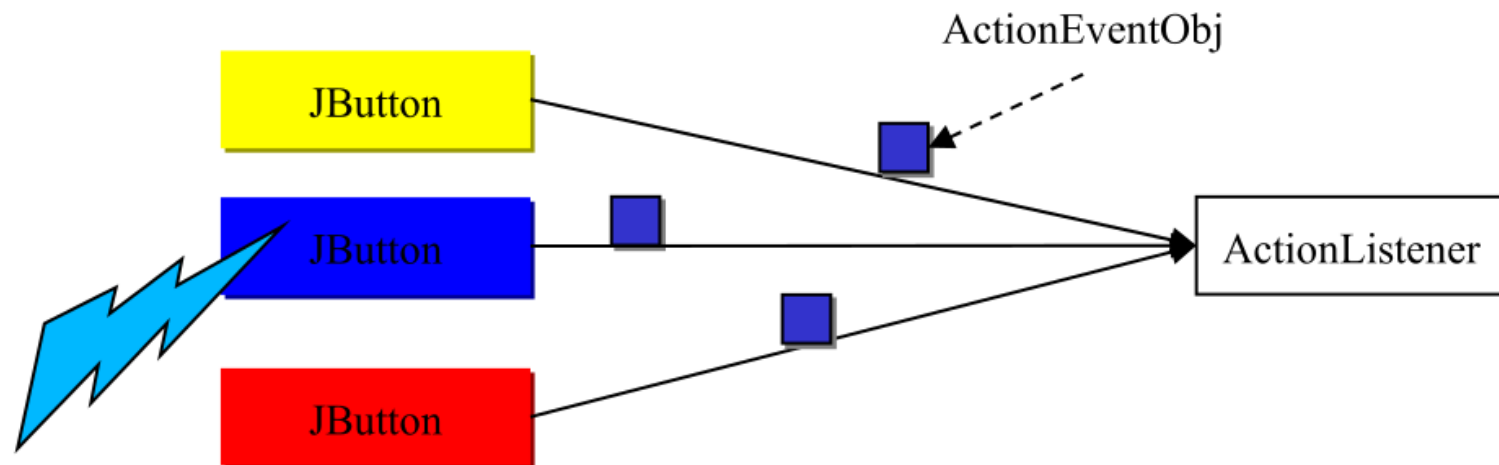
|                              |  |
|------------------------------|--|
| <b>ActionListener</b>        | User clicks a button, presses Enter while typing in a text field, or chooses a menu item |
| <b>WindowListener</b>        | User closes a frame  |
| <b>MouseListener</b>         | User presses a mouse button while the cursor is over a component                         |
| <b>MouseMotionListener</b>   | User moves the mouse over a component  |
| <b>FocusListener</b>         | Component gets the keyboard focus  |
| <b>ListSelectionListener</b> | Table or list selection changes  |

## Esempio: reazione al click su un bottone

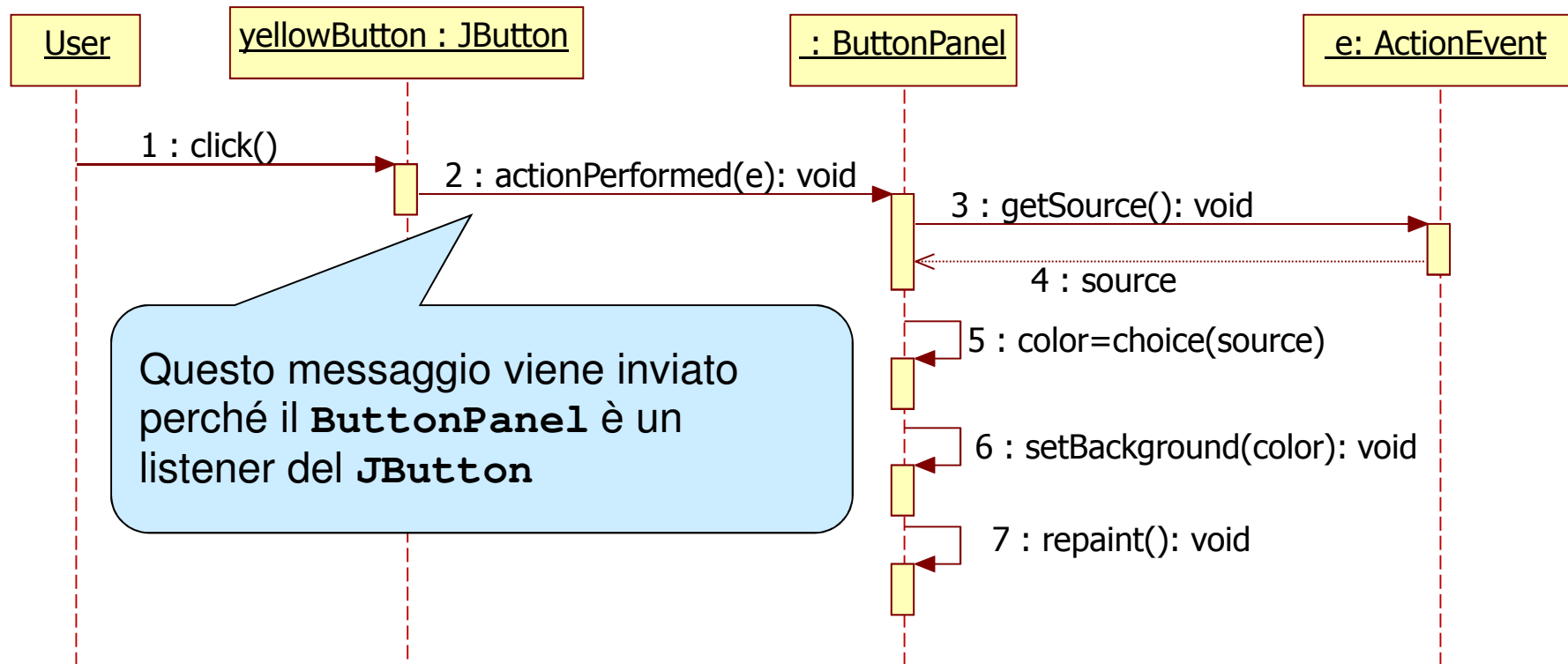
- In funzione del bottone cliccato coloriamo il panel in modo diverso.



- Creiamo un solo Listener per tutti i JButton

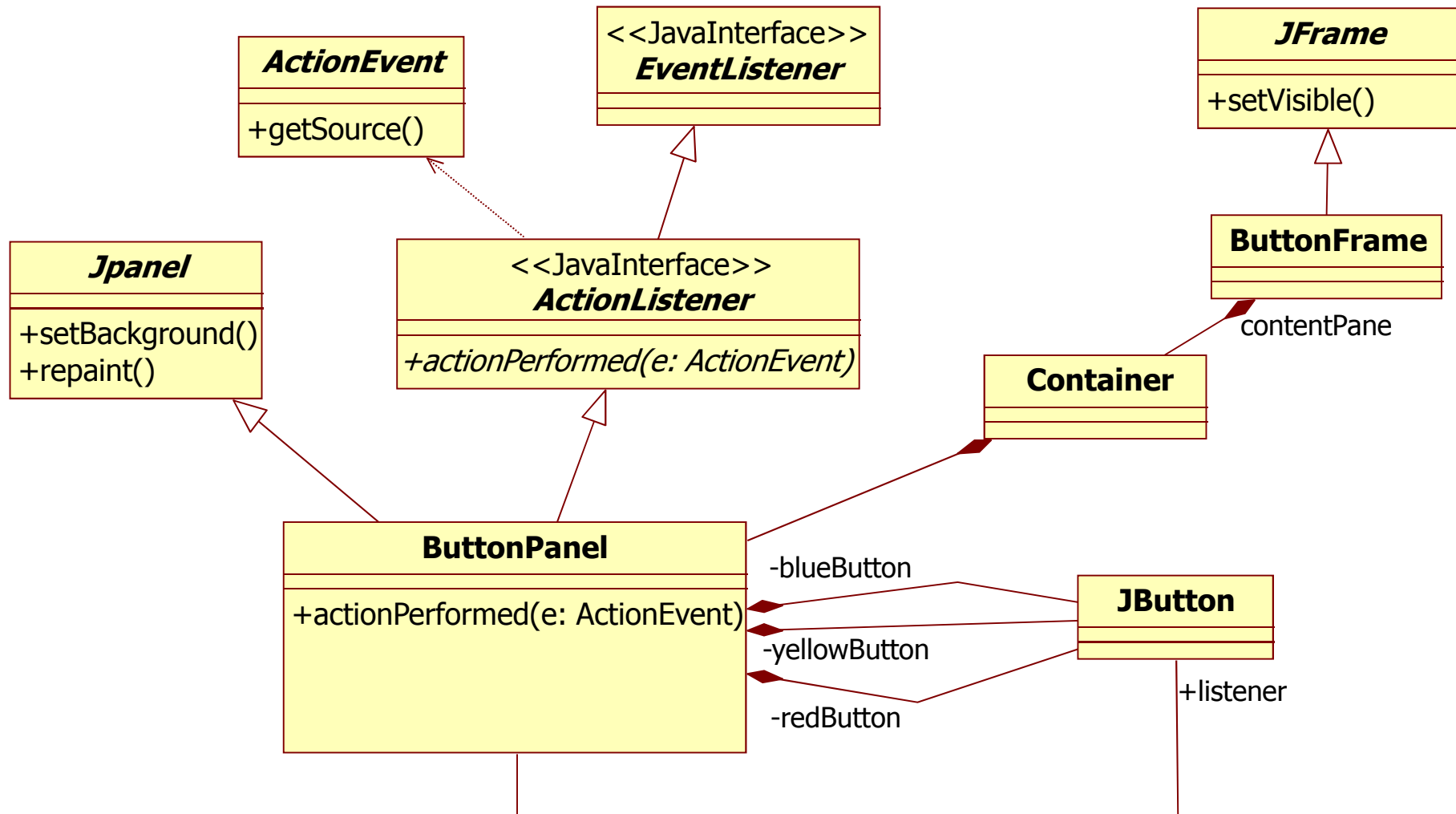


# Funzionamento desiderato





# Class diagram







## Esempio: implementazione

---

```
import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JPanel;

public class ButtonPanel extends JPanel
    implements ActionListener {
    private static final long serialVersionUID = 1L;
    JButton yellowButton;
    JButton blueButton;
    JButton redButton;
```



## Esempio: implementazione

---

```
public ButtonPanel() {  
    yellowButton = new JButton("Yellow");  
    blueButton = new JButton("Blue");  
    redButton = new JButton("Red");  
    add(yellowButton);  
    add(blueButton);  
    add(redButton);  
    yellowButton.addActionListener(this);  
    blueButton.addActionListener(this);  
    redButton.addActionListener(this);  
}
```



## Esempio: implementazione

---

```
public void actionPerformed(ActionEvent evt) {  
    Object source = evt.getSource();  
    Color color = getBackground();  
    if(source.equals(yellowButton))  
        color = Color.yellow;  
    else if (source.equals(blueButton))  
        color = Color.blue;  
    else if (source.equals(redButton))  
        color = Color.red;  
    this.setBackground(color);  
    this.repaint();  
}  
}
```



## Esempio: implementazione

---

```
import java.awt.*;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.*;
class ButtonFrame extends JFrame {
    private static final long serialVersionUID = 1L;
    public ButtonFrame() {
        setTitle("ButtonTest"); setSize(300, 200);
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
        Container contentPane = getContentPane();
        contentPane.add(new ButtonPanel());
    }
}
```



## Esempio: implementazione

---

```
import javax.swing.JFrame;
public class ButtonTest {
    public static void main(String[] args) {
        JFrame frame = new ButtonFrame();
        frame.setVisible(true);
    }
}
```



---

# IL PATTERN OBSERVER



## Il pattern Observer

---

- Conosciuto anche come: Publish-Subscribe, Callback, Dependents
- È un pattern comportamentale che gestisce la **comunicazione** tra oggetti
- Definisce il modo in cui un certo numero di classi possono ricevere **notifiche di eventi** cui sono interessate
- Il pattern Observer definisce una dipendenza uno-a-molti fra oggetti in modo che quando un oggetto cambia stato, tutti gli oggetti interessati ne sono informati



# Motivazioni per il Pattern Observer

---

- Si immagini un programma che gestisce una biblioteca.
  - ▶ L'applicazione gestisce i prestiti e le ricerche.
- Se un utente restituisce un libro, gli utenti interessati a quel libro dovranno ricevere una notifica del cambiamento di disponibilità del libro.
- L'alternativa sarebbe il polling: chi è interessato a un libro che non era disponibile l'ultima volta che lo ha cercato interroga periodicamente il sistema per verificare se ci sono state variazioni della situazione.
  - ▶ Poco efficiente (un sacco di interrogazioni vengono fatte a vuoto)
  - ▶ Poco efficace (il libro potrebbe diventare disponibile un microsecondo dopo che ho fatto l'interrogazione)





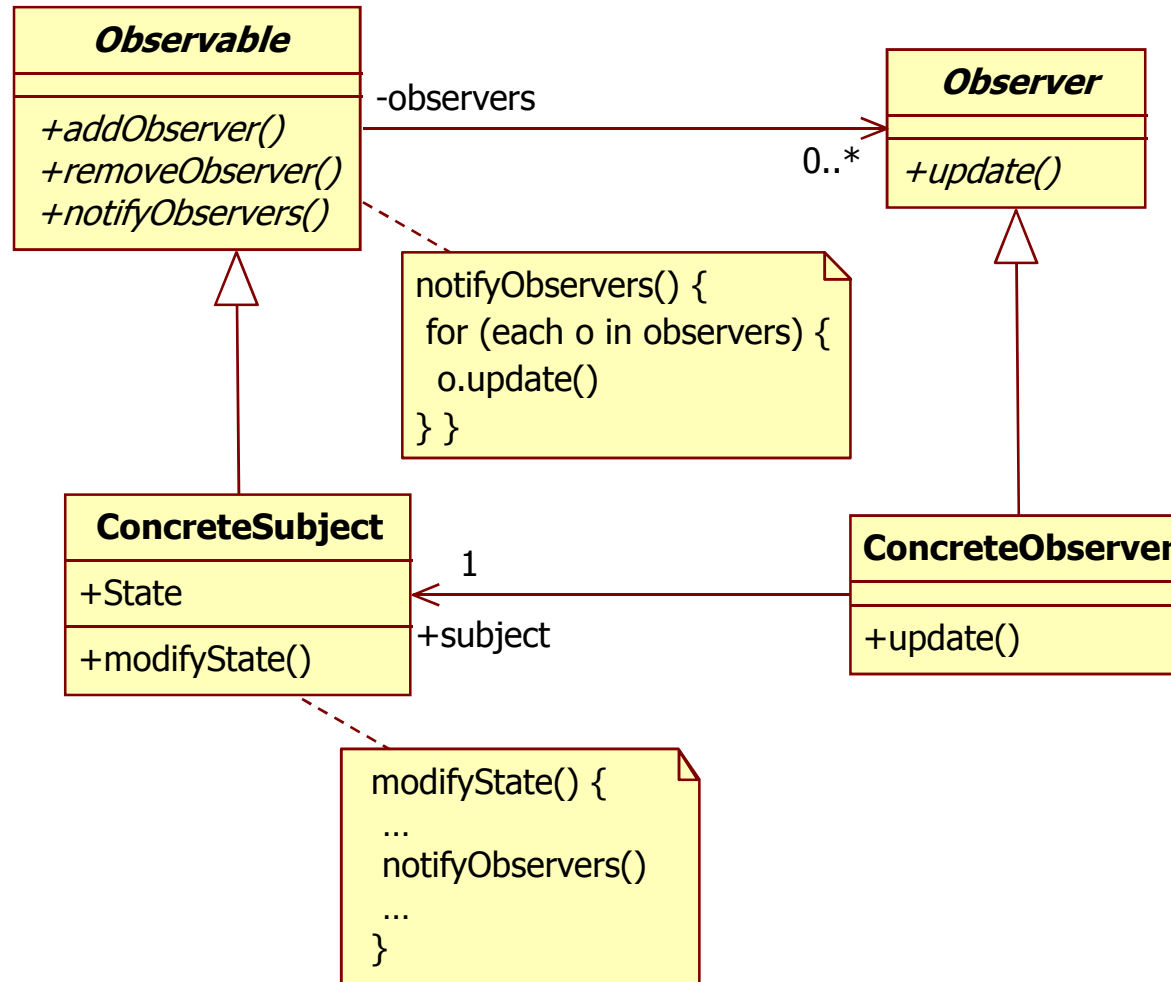
## Pattern Observer: i partecipanti

---

- Il soggetto osservabile, dove accadono gli eventi
- Gli osservatori, che ricevono notifica degli eventi

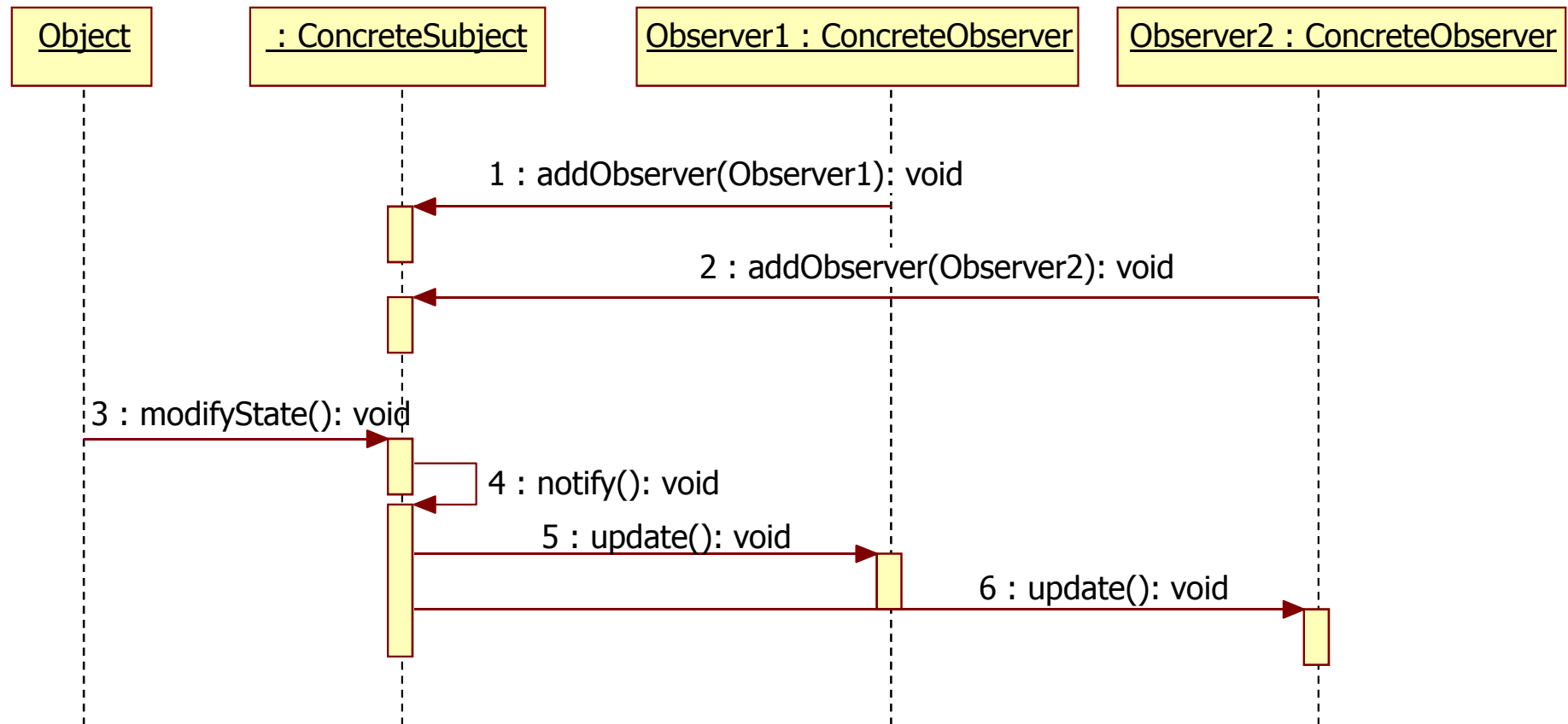


# Pattern Observer





# Pattern Observer





# Interface Observer

---

```
public interface Observer {  
    public void update(Observable o, Object arg);  
}
```



# L'interfaccia Observer

---

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

- Questo metodo viene chiamato dall'oggetto **Observable** associato, allo scopo di notificare un evento all'observer.
  - ▶ quando viene chiamato il metodo **notifyObservers** dell'oggetto **Observable** associato.
  - ▶ è possibile che l'oggetto **Observable** chiami lui stesso **notifyObservers** a fronte di un cambiamento nel proprio stato.
- Parametri:
  - ▶ o – l'oggetto **Observable**.
  - ▶ arg – l'argomento passato a **notifyObservers**.



# Class Observable

---

```
import java.util.*;

public class Observable {
    protected List<Observer> observers =
        new ArrayList<Observer>();

    public void addObserver(Observer o) {
        observers.add(o);
    }

    public void removeObserver(Observer o) {
        observers.remove(o);
    }

    protected void notifyObservers(Object obj) {
        for(Observer o : observers) {
            o.update(this, obj);
        }
    }
}
```



## Esempio

---

- Il soggetto concreto osservato gestisce input da tastiera.
- L'input di ogni riga è considerato un evento, che deve essere notificato agli osservatori concreti.
- Ogni volta che si legge una stringa da **System.in** si chiama **notifyObservers**, per informare tutti gli osservatori dell'evento di lettura.



## NB

---

- Java forniva delle classi di libreria per implementare il pattern observer.
- Quelle classi sono poi state deprecate.
- Questo non vuol dire che il pattern non si debba usare!
- Nel seguito usiamo la nostra implementazione delle classi observer e observable.
- Sul sito dell'e-learning e' ancora possibile trovare il codice che utilizza le classi deprecate
  - ▶ Nelle pagine del corso relative agli anni scorsi.





# Implementazione dell'osservatore concreto

---

```
public class InputHandler implements Observer {  
    private String info;  
    public void update(Observable o, Object arg){  
        this.info = (String) arg;  
        System.out.println(info + " communicated");  
    }  
}
```



# Implementazione del soggetto concreto

---

```
public class EventSource extends Observable
                                implements Runnable {
    public void run() {
        try {
            final InputStreamReader isr =
                                new InputStreamReader(System.in);
            final BufferedReader br = new BufferedReader(isr);
            while(true) {
                System.out.println("Enter Text >");
                final String str = br.readLine();
                notifyObservers(str);
            }
        }
        catch (IOException e) {}
    }
}
```

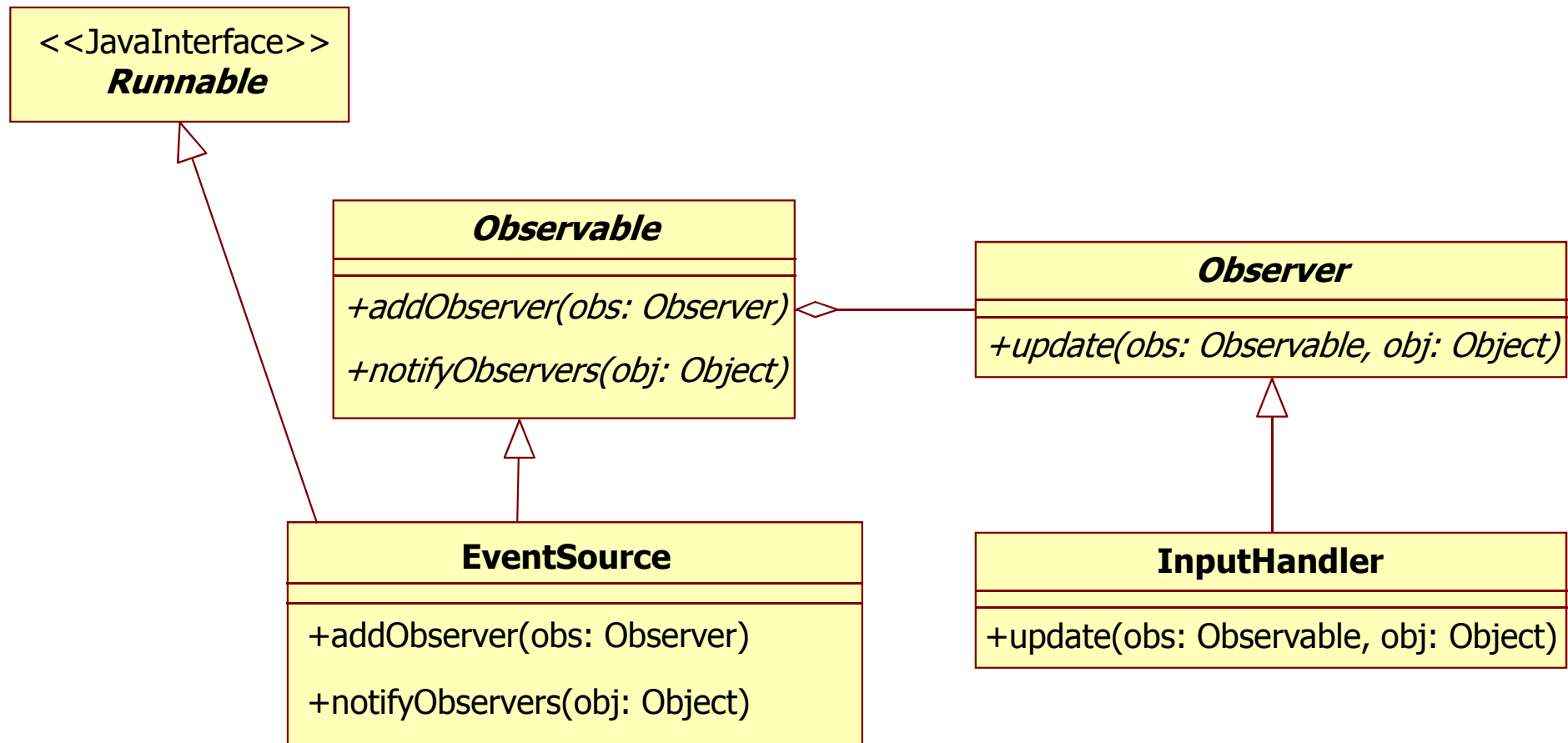


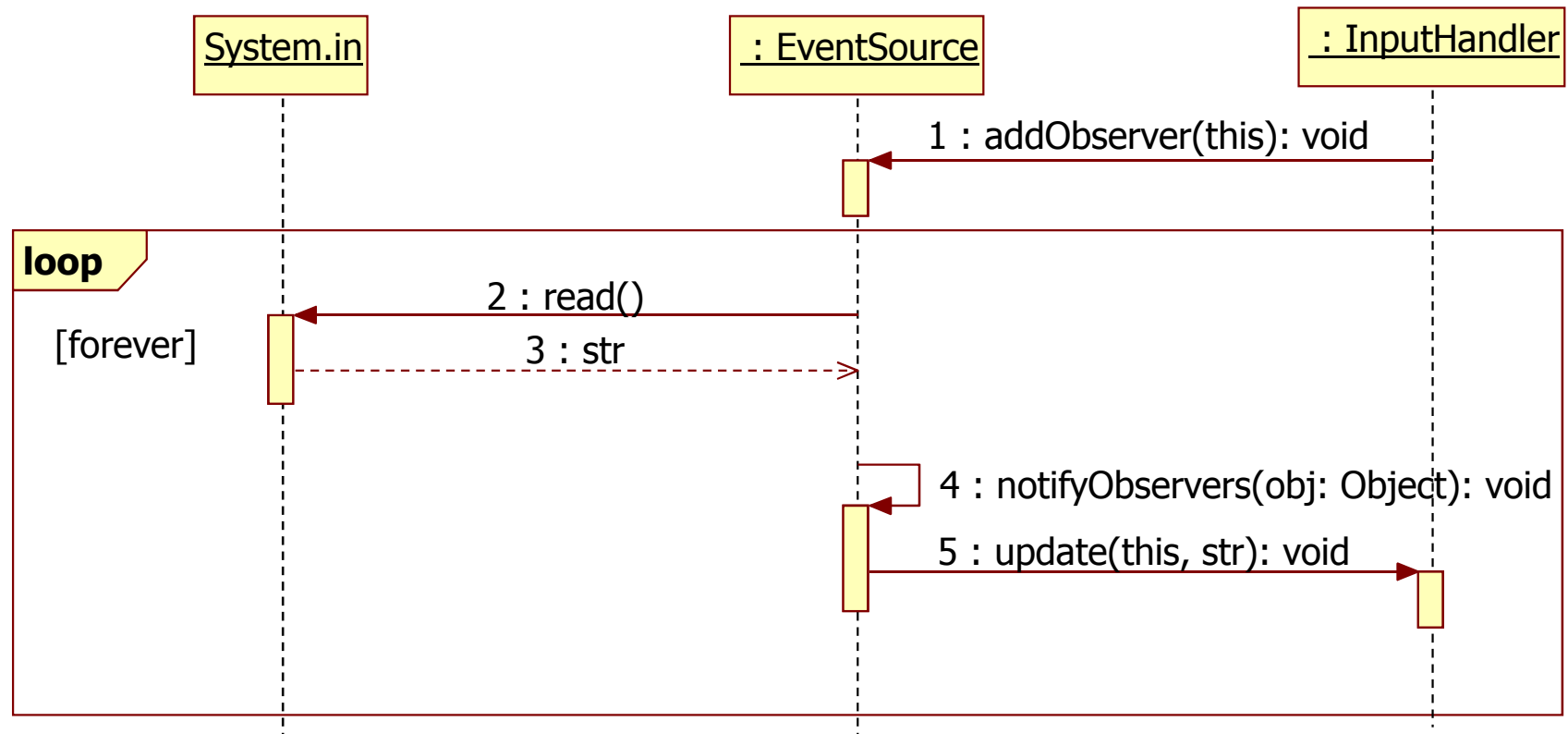
# Applicazione

---

```
public class myapp {  
    public static void main(String args[]) {  
        // create an event source - reads from stdin  
        final EventSource evSrc = new EventSource();  
        // create an observer  
        final InputHandler inHandler = new InputHandler();  
        // subscribe the observer to the event source  
        evSrc.addObserver(inHandler);  
        // starts the event thread  
        Thread evSrcThread = new Thread(evSrc);  
        evSrcThread.start();  
    }  
}
```

NB: in questo caso non è utile creare un thread (il main finisce subito dopo).  
Ma se creassimo tanti thread, ciascuno potrebbe mandare le sue notifiche all'osservatore. Quindi l'osservatore potrebbe essere informato su tanti fenomeni concorrenti.







# Osservazione

---

- La situazione vista ricorre piuttosto frequentemente
  - ▶ Indipendentemente dall'uso del pattern observer
- C'è un programma che fa quel che deve fare e intanto un thread si preoccupa di gestire eventi (spesso si tratta di input)



---

# **RMI CALLBACK CON PATTERN OBSERVER**



## Esempio: Timer

---

- Creiamo un server che invia al client una notifica ogni X secondi.
- Il client indica al server ogni quanti secondi desidera ricevere la notifica.
- Usiamo il pattern Observer:
  - ▶ Il server è un **RemoteObservable**
  - ▶ Il client si registra come **RemoteObserver**
- Cominciamo a realizzare la soluzione per un solo client, poi la estendiamo al caso con più client.







# Interfaccia RemoteObserver

---

```
import java.rmi.Remote;  
import java.rmi.RemoteException;  
  
public interface RemoteObserver extends Remote {  
    void remoteUpdate(Object observable, Object arg)  
        throws RemoteException;  
}
```



# Interfaccia RemoteObservable

---

```
import java.rmi.*;

public interface RemoteObservable extends Remote {
    public void addObserver(RemoteObserver o)
                                throws RemoteException;
    public void removeRemoteObserver(RemoteObserver o)
                                throws RemoteException;
    public void notifyRemoteObservers(Object obj)
                                throws RemoteException;
}
```



## Interfaccia `TimeTickService`

---

```
import java.rmi.RemoteException;

public interface TimeTickService extends RemoteObservable {
    void setTick(int period) throws RemoteException;
}
```



## Classe Client

---

```
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

public class Client extends UnicastRemoteObject
    implements RemoteObserver {
    private static final long serialVersionUID = 1L;
    Client() throws RemoteException {    }
    public void remoteUpdate(Object observable, Object arg)
        throws RemoteException {
        System.out.println("got message:" + arg);
    }
}
```



## Classe Client

---

```
public static void main(String[] args) {  
    try {  
        Registry registry = LocateRegistry.getRegistry(1099);  
        TimeTickService remoteService =  
            (TimeTickService) registry.lookup("TimeTick");  
        RemoteObserver client = new Client();  
        remoteService.addRemoteObserver(client);  
        remoteService.setTick(5);  
    } catch (Exception ex) {  
        ex.printStackTrace();  
    }  
}
```



## Classe TimeTickServer

---

```
import java.util.*;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;

public class TimeTickServer
    implements RemoteObservable, TimeTickService {
    private List<RemoteObserver> observers =
        new ArrayList<RemoteObserver>();
    private int tickPeriod=-99;
    public TimeTickServer() { }
```



## Classe `TimeTickServer`

---

```
public void setTick(int period) throws RemoteException {
    tickPeriod = period;
}

public void addRemoteObserver(RemoteObserver o)
                               throws RemoteException {
    observers.add(o);
    System.out.println("Added observer:" + o);
}

public void removeRemoteObserver(RemoteObserver o)
                                  throws RemoteException {
    observers.remove(o);
}
```





## Classe `TimeTickServer`

---

```
public void notifyRemoteObservers(Object obj)
                                   throws RemoteException {
    RemoteObserver o=null;
    int numObservers=observers.size();
    int i=0;
    while(i<numObservers) {
        o=observers.get(i);
        try {
            o.remoteUpdate(this, obj);
            i++;
        } catch (RemoteException e) {
            observers.remove(o);
            if ((--numObservers)==0) {
                tickPeriod=-99;
            }
        }
    }
}
```

---



## Classe TimeTickServer

---

```
private void exec() {  
    long target, now;  
    try {  
        TimeTickService stub = (TimeTickService)  
            UnicastRemoteObject.exportObject(this, 3939);  
        Registry reg = LocateRegistry.createRegistry(1099);  
        reg.rebind("TimeTick", stub);  
        System.err.println("Server ready");  
        while(true) {  
            target=1000*tickPeriod+System.currentTimeMillis();  
            while(tickPeriod<0 ||  
                (now=System.currentTimeMillis())<target) {  
                Thread.sleep(500);  
            }  
            notifyRemoteObservers(now);  
        }  
    } catch (Exception ex) { }  
}
```



## Classe `TimeTickServer`

---


```
public static void main(String[] args) {  
    TimeTickServer obj = new TimeTickServer();  
    obj.exec();  
}  
}
```





# Deploy


---

## ▼ Client


 Client.java


 RemoteObservable.java

 RemoteObserver.java


 TimeTickService.java

## ▼ Server

 RemoteObservable.java

 RemoteObserver.java

 TimeTickServer.java

 TimeTickService.java



## Limiti dell'implementazione vista

---

- Tutto funziona a dovere finché abbiamo un solo client.
- Ma se ci sono diversi client e ciascuno desidera ricevere una notifica con un periodo diverso, occorre che ciascun client venga gestito ad hoc



# Sistema multi-client

---

- Un problema sta nel fatto che ciascun client può richiedere di ricevere notifiche con una frequenza diversa da quella usata per gli altri client.
- Soluzione:
  - ▶ Costruiamo una struttura dati che non contiene solo i riferimenti (remoti) ai client, ma anche le caratteristiche del servizio richiesto
    - Periodo
    - Quanto tempo è passato dall'ultima notifica
  - ▶ Il server con frequenza fissa (ad es. ogni secondo) va a incrementare il tempo passato dall'ultima notifica per ogni client. Ai client per cui il tempo trascorso è diventato pari al periodo viene mandata la notifica e si resetta il tempo trascorso
    - La notifica si manda con la solita update del pattern observer



# Interfaccia RemoteObserver

---

```
import java.rmi.Remote;  
import java.rmi.RemoteException;  
  
public interface RemoteObserver extends Remote {  
    void remoteUpdate(Object observable, Object arg)  
        throws RemoteException;  
}
```



# Interfaccia RemoteObservable

---

```
import java.rmi.*;

public interface RemoteObservable extends Remote {
    public void addRemoteObserver(RemoteObserver o)
                                   throws RemoteException;
    public void removeRemoteObserver(RemoteObserver o)
                                   throws RemoteException;
    public void notifyRemoteObservers(Object obj)
                                   throws RemoteException;
}
```





# Interfaccia RemoteObserver

---

```
import java.rmi.Remote;  
import java.rmi.RemoteException;  
  
public interface RemoteObserver extends Remote {  
    void remoteUpdate(Object observable, Object arg)  
        throws RemoteException;  
}
```



## Interfaccia `TimeTickService`

---

```
import java.rmi.RemoteException;

public interface TimeTickService
    extends RemoteObservable {
    void setPeriod(RemoteObserver o, int period)
        throws RemoteException;
}
```



## Classe TimePair

---

```
public class TimePair {
    int period, elapsed;
    TimePair(int p) {
        period=p;  elapsed=0;
    }
    public int getPeriod() { return period; }
    public void setPeriod(int period) {
        this.period = period;
    }
    public int getElapsed() { return elapsed; }
    public void setElapsed(int elapsed) {
        this.elapsed = elapsed;
    }
    public void incElapsed() {
        this.elapsed++;
    }
}
```



## Classe TimeTickServer

---

```
import java.util.*;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;

public class TimeTickServer implements RemoteObservable,
                                         TimeTickService {
    private Hashtable<RemoteObserver, TimePair> observers =
        new Hashtable<RemoteObserver, TimePair>();
    public TimeTickServer() {
    }
}
```



## Classe `TimeTickServer`

---

```
public synchronized void
    addRemoteObserver (RemoteObserver o)
                                throws RemoteException {
    observers.put(o, new TimePair(-99));
    System.out.println("Added observer:" + o);
}

public synchronized void
    removeRemoteObserver (RemoteObserver o)
                                throws RemoteException {
    observers.remove(o);
}

public synchronized void setPeriod(RemoteObserver o,
                                int period)
                                throws RemoteException {
    if(observers.containsKey(o)) {
        observers.put(o, new TimePair(period));
    }
}
```



## Classe `TimeTickServer`

```
public synchronized void notifyRemoteObservers (
                                Object obj) {

    TimePair tp=null;
    Enumeration<RemoteObserver> keys = observers.keys();
    while (keys.hasMoreElements()) {
        RemoteObserver key = keys.nextElement();
        tp=observers.get(key);
        tp.incElapsed();
        if (tp.getElapsed()==tp.getPeriod()) {
            tp.setElapsed(0);
            observers.put(key, tp);
            try {
                key.remoteUpdate(this, tp.getPeriod());
            } catch (RemoteException e) {
                observers.remove(key);
            }
        } else {
            observers.put(key, tp);
        }
    }
}
```



## Classe TimeTickServer

---

```
private void exec() {
    TimeTickService stub;
    try {
        stub = (TimeTickService)
            UnicastRemoteObject.exportObject(this, 3939);
        Registry registry = LocateRegistry.createRegistry(1099);
        registry.rebind("TimeTick", stub);
    } catch (RemoteException e1) {
        System.err.println("Server unable to start");
        System.exit(0);
    }
    System.err.println("Server ready");
    while(true) {
        try { Thread.sleep(1000); }
        catch (InterruptedException e) { }
        notifyRemoteObservers(null);
    }
}
```



## Classe `TimeTickServer`

---

```
public static void main(String[] args) {  
    TimeTickServer obj = new TimeTickServer();  
    obj.exec();  
}  
}
```





# Deploy

---

