Listeners

Gian Enrico Conti

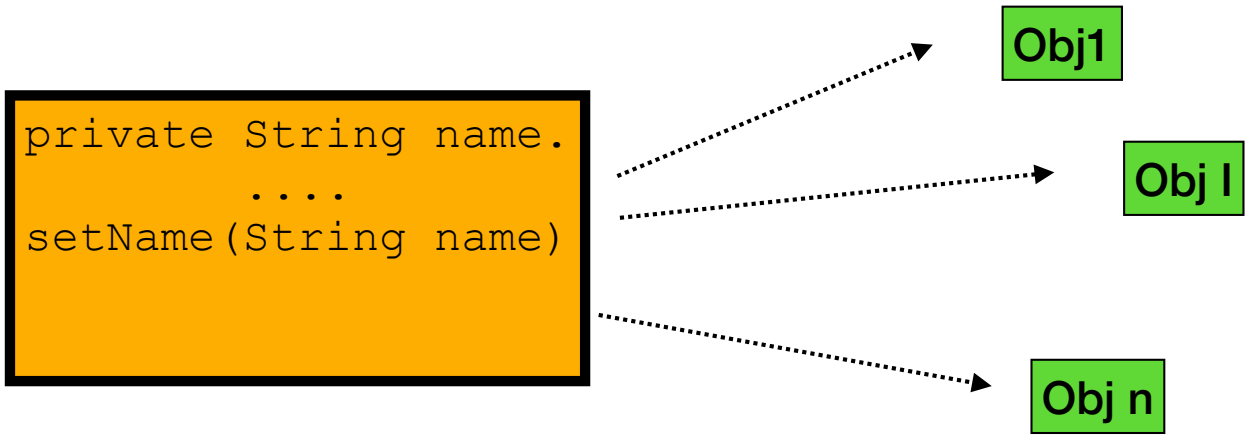
Prova Finale - Ingegneria del Software - AA 20xx/20xy

Intro

Vogliamo un meccanismo che permetta a PIU oggetti di essere notificati su un cambiamento.

Perchè?

a) Per esempio vogliamo poter aggiungere **ascoltatori senza modificare il codice della sorgente** delle modifiche.



Sul "set" notifico.

B) non è necessario tenere una lista dei potenziali "interessati alle modifiche"

C) disaccoppiare le modifiche al modello dalle view che devono mostrare le modifiche.

Apis

Java8

Ci servirà:

```
PropertyChangeListener listener;
 // https://docs.oracle.com/javase/8/docs/api/java/beans/PropertyChangeListener.html
```

Se l' oggetto su cui in ascolto cambia, riceverà notifica del cambio.

Andrà associato / registrato sull' oggetto che CAMBIA!

Il listener verrà "avvisato" tramite callback:

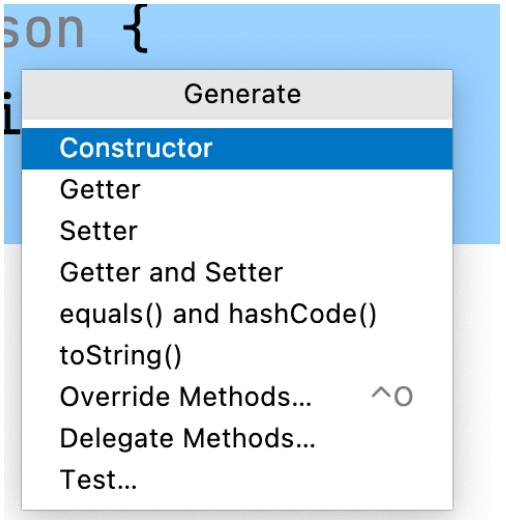
void propertyChange(PropertyChangeEvent evt)

Ora x gradi...

Esempio: notifica del cambio di prezzo di un oggetto Product:

Code:

```
public class Product {
    private String name;
    private double cost;
}
```



Solito

Setter, getter, constructor...

Esempio: notifica del cambio di prezzo di un oggetto Product: listener

Code:

```
public class Product {
    private String name;
    private double cost;

    PropertyChangeListener listener;

    public Product(String name, double cost) {
        this.name = name;
        this.cost = cost;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public double getCost() {
        return cost;
    }

    public void setCost(double cost) {
        this.cost = cost;
    }

    public void setListener(PropertyChangeListener listener) {
        this.listener = listener;
    }
}
```

Nel main:

```
public class App
{
    public static void main( String[] args )
    {
        System.out.println( "Hello World!" );
        Product p = new Product("apple", 1);
        p.setListener(..);

    }
}
```

Ci serve un listener. P.es. simuliamo che venga mandata MAIL:

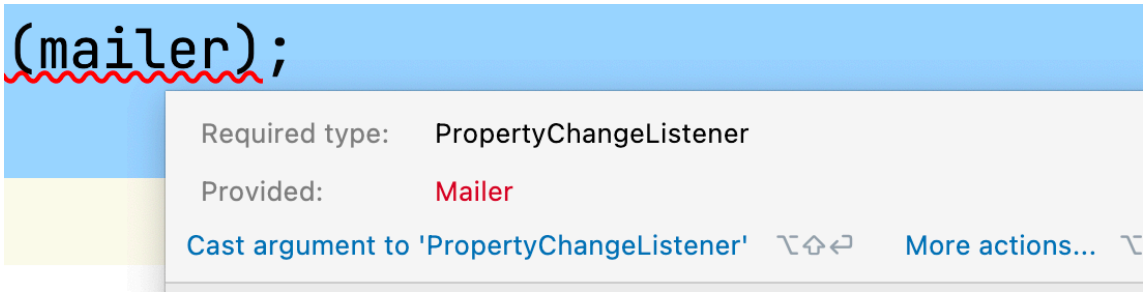
```
public class Mailer {

    void sendMsg(String msg){

    }
}
```

Nel MAIN:

```
public class App
{
    public static void main( String[] args )
{
    System.out.println( "Hello World!" );
    Product p = new Product("apple", 1);
    Mailer mailer = new Mailer();
    p.setListener(mailer);
}
```



Non implementa la interfaccia.
Facciamolo:

```
public class Mailer implements PropertyChangeListener {

...
```

IntelliJ suggerisce i metodi required:

```
public class Mailer implements PropertyChangeListener {

    void sendMsg(String msg){

    }

    @Override
    public void propertyChange(PropertyChangeEvent evt) {

    }
}
```

Mettiamo 2 stampe x debug:

metodi:

```
public class Mailer implements PropertyChangeListener {

    void sendMsg(String msg){
        System.out.println("sending  " + msg);
    }

    @Override
    public void propertyChange(PropertyChangeEvent evt) {
        System.out.println("evt  " + evt);
    }
}
```

Nel MAIN, per mostrare che avviene:

Main:

```
public static void main( String[] args )
{
    System.out.println( "Hello World!" );
    Product p = new Product("apple", 1);
    Mailer mailer = new Mailer();
    p.setListener(mailer);
    p.setCost(100);
}
```

NON succede nulla..

Modifica al SETTER:

Manca la invocazione dei metodi necessari:

```
void propertyChange(PropertyChangeEvent evt);
```

E faremo:

```
public void setCost(double cost) {
    this.cost = cost;

    PropertyChangeEvent evt = PropertyChangeEvent(......)
    this.listener.propertyChange(evt);
}
```

...

Ma serve un evento, che si crea con:

```
public PropertyChangeEvent(Object source,
                    String propertyName,
                Object oldValue,
                    Object newValue)
```

Evento:

Creiamo evento:

```
  PropertyChangeEvent evt = new PropertyChangeEvent(
            this,
    "COST_CHANGED",
            this.cost,
            cost);
    this.cost = cost;

    this.listener.propertyChange(evt);
}
```

Quindi avremo:

Setter e evento:

```
public void setCost(double cost) {

    PropertyChangeEvent evt = new PropertyChangeEvent(
            this,
    "COST_CHANGED",
            this.cost,
            cost);
    this.cost = cost;

    this.listener.propertyChange(evt);
}
```

SI NOTI OLD / NEW

RUN...

Run:

```
CONSOLE:

/Users..... org.example.App
Hello World!

evt  java.beans.PropertyChangeEvent[propertyName=COST_CHANGED;
oldValue=1.0; newValue=100.0; propagationId=null;
source=org.example.Product@2ff4acd0]
```

OK!

Leggera modifica al codice di listening:

Mandiamo messaggio:

```
public void propertyChange(PropertyChangeEvent evt) {
    //System.out.println("evt  " + evt);
    sendMsg(evt.getPropertyName() +
            " from " +
            evt.getOldValue() +
            " to " +
            evt.getNewValue()
    );
}
```

RUN...

```
Hello World!
sending  COST_CHANGED from 1.0 to 100.0
```

Potremmo anche passare lo stesso oggetto (copia !?) o un oggetto apposito.

On timer…

Per darne evidenza anche su asynch:

```
..
    p.setListener(mailer);
    buidMyTimer(p);
    //was p.setCost(100);
}

static void buidMyTimer(){
....
```

On timer...

```
static double fakeCost = 100;

static void buidMyTimer(final Product p){

    TimerTask task = new TimerTask() {
        public void run() {
            p.setCost(fakeCost++);
        }
    };
    Timer timer = new Timer("Timer");

    long delay = 1000L;
    timer.scheduleAtFixedRate(task, 0,1000);
}


>RUN....
```