Observer

1. We need to be şnformed when certain things happen
   1. Objects field changes
   2. Object does something
   3. Some external event occurs
2. We want to listen to events and notified when they occur
3. Typical pattern involves addXXXListener()
4. Java now has functional objects
   1. Supplier<T>, Consumer<T>, Function<T>
5. Observer: An observer is an object that wishes to be informed about events happening in the system The entity generating the events is an observable.

Observer and Observable

1. In the core you have observers that observe observable objects and are notified when there are some changes.
2. Example code is important.

An event class

1. Example code!!
2. Same logic with event based approach
3. No need for the class to implement or extend observables and observers.

The problem with dependent properties

1. Example code: Can vote concept is dependent on age.

package com.activemesa.behavioral.observer.events;

import java.util.\*;

import java.util.function.Consumer;

class Event<TArgs>

{

private int count = 0;

private Map<Integer, Consumer<TArgs>>

handlers = new HashMap<>();

public Subscription addHandler(Consumer<TArgs> handler)

{

int i = count;

handlers.put(count++, handler);

return new Subscription(this, i);

}

public void fire(TArgs args)

{

for (Consumer<TArgs> handler : handlers.values())

handler.accept(args);

}

public class Subscription implements AutoCloseable

{

private Event<TArgs> event;

private int id;

public Subscription(Event<TArgs> event, int id)

{

this.event = event;

this.id = id;

}

@Override

public void close() /\*throws Exception\*/

{

event.handlers.remove(id);

}

}

}

class PropertyChangedEventArgs

{

public Object source;

public String propertyName;

public PropertyChangedEventArgs(Object source, String propertyName)

{

this.source = source;

this.propertyName = propertyName;

}

}

class Person

{

public Event<PropertyChangedEventArgs>

propertyChanged = new Event<>();

private int age;

public int getAge()

{

return age;

}

public void setAge(int age)

{

if (this.age == age) return;

boolean oldCanVote = getCanVote();

this.age = age;

propertyChanged.fire(new PropertyChangedEventArgs(

this, "age"

));

if (oldCanVote != getCanVote())

{

propertyChanged.fire(new PropertyChangedEventArgs(

this, "canVote"

));

}

}

public boolean getCanVote()

{

return age >= 18;

}

}

class HandmadeEventsDemo

{

public static void main(String [] args)

{

Person person = new Person();

Event<PropertyChangedEventArgs>.Subscription sub =

person.propertyChanged.addHandler(x -> {

System.out.println("Person's "

+ x.propertyName + " has changed");

});

person.setAge(17);

person.setAge(18);

sub.close();

person.setAge(19);

}

}



Summary

1. Observer is an intrusive approach: an observable must provide an event to subscribe to
2. Special care must be taken to prevent issues in multithreaded scenarios
3. Rx uses Observer<T> / Oberable<T>