Java Listener inheritance

Asked 16 years ago Modified 15 years, 10 months ago Viewed 10k times

7

I have a java class which fires custom java events. The structure of the code is the following:

```
public class AEvent extends EventObject {
...
}

public interface AListener extends EventListener {

  public void event1(AEvent event);

}

public class A {

  public synchronized void addAListener(AListener l) {
  ..
  }

  public synchronized void removeAListener(AListener l) {
  ..
  }

  protected void fireAListenerEvent1(AEvent event) {
  ..
  }
}
```

Everything works correctly, but I'd like to create a new subclass of A (call it B), which may fire a new event. I'm thinking of the following modification:

```
public class BEvent extends AEvent {
...
}

public interface BListener extends AListener {

  public void event2(BEvent event);
}

public class B extends A {

  public synchronized void addBListener(BListener l) {
  ..
  }

  public synchronized void removeBListener(BListener l) {
  ..
  }
  protected void fireBListenerEvent2(AEvent event) {
  ..
  }

}
```

Is this the correct approach? I was searching the web for examples, but couldn't find any.

There are a few things I don't like in this solution:

1. `BListener` has two methods one uses `AEvent` the other uses `BEvent` as a parameter.
2. `B` class both has `addAListener` and `addBListener` methods. Should I hide addAListener with private keyword? **[UPDATE: it's not possible to hide with private keyword]**
3. Similar problem with `fireAListenerEvent1` and `fireBListenerEvent1` methods.

I'm using Java version 1.5.

java inheritance events listener

Share
Improve this question
Follow

edited Dec 16, 2008 at 12:57

asked Dec 16, 2008 at 8:17
asalamon74
6,170 ● 9 ● 48 ● 60

Could you expand a bit on your motives for this design? Why does matter that B extend A, in what kind of situations are all this stuff going to be used? Otherwise uzhin's approach seems to be pointing in a right direction. – Slartibartfast Jan 27, 2009 at 14:27

B adds new functionality to class A. It means that the event fired by A (event1) is a valid event for class B. Because of the new functionality a new event (event2) is also valid for B. I'd like to handle the two events (event1+event2) together since they are related. – asalamon74 Jan 27, 2009 at 15:14

Side note: a ConcurrentModificationException can occur in the fireXXX methods unless you synchronize those as well (expensive!). For typical use cases, it's faster and less error-prone to use java.util.concurrent.CopyOnWriteArrayList for your listener list so that synchronization is unnecessary. – flicken Jan 29, 2009 at 13:59

6 Answers

Sorted by: Highest score (default)

I don't see a reason why `BListener` should extend `AListener`.

13

Do you really want to force everyone interested in `B` events to also implement `event1()`?

Also you can't add `addAListener()`, since a derived class can not reduce the visibility of a method that's present in the parent class. Also, you shouldn't need to, or you would violate the Liskov substitution principle (every B must be able to do everything an A can do).

+150

And as a last remark, I'd make the `fire*()` methods protected. There's usually no reason at all to keep them public and reducing the number of public members keeps your public interface clean.

Share Improve this answer Follow

answered Dec 16, 2008 at 8:30
Joachim Sauer
308k ● 59 ● 565 ● 620

Thanks for your remarks, I've modified the question. Yes, I'd like to force everyone to implement event1 (if possible). B adds functionality to A, and B will not only fire event2 but also event1. – asalamon74 Dec 16, 2008 at 13:01

But why? Who tells you that everyone who's interested in event2 is at the same time interested in event1? When you add listener support to a component you usually do not know what it's going to be used for, so you should not force this combination. – Joachim Sauer Dec 16, 2008 at 15:48

6

Don't use inheritence, it's not what you want and will lead to a brittle and difficult to change design. Composition is a more flexible and a better approach for the design. Always try to design interfaces as granular as possible because they should not be changed event. They are your contract with the rest of the system. If new functionality needs to be added the first option is to add more information to the event. If that's not appropriate, then you should design a new interface for delivering that event. This prevents having to change any existing code which isn't affected.

Here's my favorite pattern for this, I beleive it's commonly referred to as an Observer.

Make a new interface defining a methods for that event type (fooEvent() addFooEventListener() removeFooEventListener()). Implement this interface in the concrete class which generates these events. (I usually calls this something like SourcesFooEvent, FiresFooEvent, FooEventSource, etc)

If you want to reduce code duplication you can construct a helper class which handles registration of the listeners, stores them in a collection, and provides a fire method for publishing the events.

Generics can help here. First, a generic listener interface:

```
public interface Listener<T> {
  void event(T event);
}
```

Next, a matching EventSource interface:

```
public interface EventSource<T> {
    void addListener(Listener<T> listener);
}
```

Finally an abstract base class to quickly construct a helper class to handle registration of listeners and event dispatch:

```
public abstract class EventDispatcher<T> {
    private List<Listener<T>> listeners = new CopyOnWriteArrayList<T>();

    void addListener(Listener<T> listener) {
      listeners.add(listener);
    }

    void removeListener(Listener<T> listener) {
      listeners.remove(listener);
    }

    void fireEvent(T event) {
      for (Listener<T> listener : listeners) {
        listener.event(event);
      }
    }
}
```

You'd make use of the abstract EventDispatcher through encapsulation, allowing any other class to easily implement EventSource while not requiring it to extend any particular class.

```
public class Message {
}

public class InBox implements EventSource<Message> {

  private final EventDispatcher<Message> dispatcher = new
EventDispatcher<Message>();

  public void addListener(Listener<Message> listener) {
    dispatcher.addListener(listener);
  }

  public void removeListener(Listener<Message> listener) {
    dispatcher.removeListener(listener);
  }

  public pollForMail() {
    // check for new messages here...
    // pretend we get a new message...

    dispatcher.fireEvent(newMessage);
  }
}
```

Hopefully this illustrates the nice balance between type safety (important), flexibility and code reuse.

Share

Improve this answer

Follow

edited Feb 1, 2009 at 15:42

answered Jan 29, 2009 at 15:54

Mark Renouf

31k ● 19 ● 96 ● 125

If you're going to use Java 5 features, then you should use a CopyOnWriteArrayList instead of an ArrayList for EventDispatcher<T> to avoid concurrency problems – Eddie Feb 1, 2009 at 0:37

Mark, I think you might want new CopyOnWriteArrayList<Listener<T>>(); instead. The code doesn't compile otherwise. – Chris Andrews Jan 8, 2014 at 19:53

3

I understand from your comment to saua that firing B will automatically fire A.

Why not use a single type of listener and then mix some inheritance, delegation and generics?

```
class AEvent {}
class BEvent extends Event{}

interface EventListner<E extends AEvent>
{
   onEvent(E e);
}

class ListenerManager<E extends AEvent>{
    addListner(EventListener<? extends E>){}
    removeListner(EventListener<? extends E>){}
    fire(E e);
}

class A extends ListenerManager<AEvent>
{
}

class B extends ListenerManager<BEvent>
{
   A delegatorA;

  @Override addListener(EventListener<? extends BEvent> l)
  {
    super.addListner(l);
    delegatorA.addListener(l);
  }
  @Override removeListener(EventListener<? extends BEvent> l)
  {
    super.removeListner(l);
    delegatorA.removeListener(l);
  }

  @Override fire(BEvent b)
  {
    super.fire(b);
    a.fire(b)
  }

}
```

Explanation: the code for managing listeners is shared, in base class Listener Manager. B can only receive BListeners because of generics compile-time checking. Firing B will automatically fire A.

Share Improve this answer Follow

answered Dec 16, 2008 at 13:42

Yoni Roit
28.6k ● 7 ● 38 ● 32

Using generic sounds promising, but in this solution B is not a subclass of A which is quite a disadvantage. – asalamon74 Dec 16, 2008 at 13:56

They are both subclasses of ListenerManager, where all the methods are defined. Isn't it the same thing as subclassing from A? – Yoni Roit Dec 16, 2008 at 14:11

3

It seems to me you can keep things very simple.

My understading

- You have a basic class *A* that performs some *basicOperation*
- You may have a more specific subclass *B* that in addition may perform some more *specificOperations*

If that's the case, you need that both events be handled (*basic* for A and *basic* + *specific* for B)

Well, you don't need to overload the methods to do that, the only thing you need to do is add specific handlers (or listeners) for specific events.

It may be the case the event is "basic", that's fine.

But when the event is specific, you need to react accordingly. So, what I would do is to add a check **in** the specific listener to discriminate the *specific* event like this:

```
        if( whichEvent instanceof SpecificEvent ) {
            SpecificEvent s = ( SpecificEvent ) whichEvent;
            // Do something specific here...
        }
```

And that's it.

Your description of the problem is too abstract, so no concrete solutions may be suggested. Yet, if its hard to explain what you want to achieve, probably you would need to re-analyze what the problem is in first place.

If my understanding above is correct (that you need to handle *basic* + *specific* some times) the following lengthy code below may help.

Best regards

```
import java.util.*;
class A {

    // All the listener will be kept here. No matter if basic or specific.
    private List<Listener> listeners = new ArrayList<Listener>();


    public void add( Listener listener ) {
        listeners.add( listener );
    }
    public void remove( Listener listener ) {
        listeners.remove( listener );
    }


    // In normal work, this class just perform a basic operation.
    public  void normalWork(){
        performBasicOperation();
    }

    // Firing is just firing. The creation work and the
    // operation should go elsewhere.
    public void fireEvent( Event e ) {
        for( Listener l : listeners ) {
            l.eventHappened( e );
        }
    }

    // A basic operation creates a basic event
    public void performBasicOperation() {
        Event e = new BasicEvent();
        fireEvent( e );
    }
}

// Specialized version of A.
// It may perform some basic operation, but also under some special
circumstances
// it may  perform an specific operation too
```

```
class B extends A {

    // This is a new functionality added by this class.
    // Hence an specifi event is fired.
    public  void performSpecificOperation() {
        Event e = new SpecificEvent();
        // No need to fire in different way
        // an event is an event and that's it.
        fireEvent( e );
    }

    // If planets are aligned, I will perform
    // an specific operation.
    public  void normalWork(){
        if( planetsAreAligned() ) {
            performSpecificOperation();
        } else {
            performBasicOperation();
        }
    }
    private boolean planetsAreAligned() {
        //return new Random().nextInt() % 3 == 0;
        return true;
    }
}

// What's an event? Something from where you can get event info?
interface Event{
    public Object getEventInfo();
}

// This is the basic event.
class BasicEvent implements Event{
    public Object getEventInfo() {
        // Too basic I guess.
        return "\"Doh\"";
    }
}
// This is an specific event. In this case, an SpecificEvent IS-A BasicEvent.
// So , the event info is the same as its parent. "Doh".
// But, since this is an SpecificEvent, it also has some "Specific" features.
class SpecificEvent extends  BasicEvent {

    // This method is something more specific.
    // There is no need to overload or create
    // different interfaces. Just add the new  specific stuff
    public Object otherMethod() {
        return "\"All I can say is , this was an specific event\"";
    }
}

// Hey something just happened.
interface Listener {
    public void eventHappened( Event whichEvent );
}

// The basic listner gets information
// from the basic event.
class BasicEventListener implements Listener {
    public void eventHappened( Event e ) {
            System.out.println(this.getClass().getSimpleName() + ": getting
basic functionality: " + e.getEventInfo());
```

```
        }
}


// But the specific listner may handle both.
// basic and specific events.
class SpecificListener extends BasicEventListener {
    public void eventHappened( Event whichEvent ) {
        // Let the base to his work
        super.eventHappened( whichEvent );


        //  ONLY if the event if of interest to THIS object
        // it will perform something extra ( that's why it is specific )
        if( whichEvent instanceof SpecificEvent ) {
            SpecificEvent s = ( SpecificEvent ) whichEvent;
            System.out.println(this.getClass().getSimpleName() + ": aaand
getting specific functionality too: " + s.otherMethod() );
            // do something specific with s
        }
    }
}

// See it run.
// Swap from new A() to new B() and see what happens.
class Client {
    public static void main( String [] args ) {
        A a = new B();
        //A a = new A();

        a.add( new BasicEventListener() );
        a.add( new SpecificListener() );

        a.normalWork();
    }
}
```

Sample output:

```
BasicEventListener: getting basic functionality: "Doh"
SpecificListener: getting basic functionality: "Doh"
SpecificListener: aaand  getting specific functionality too: "All I can say is
, this was an specific event"
```

Taking it further, you can even get rid of the interfaces to keep it simpler

Share

Improve this answer

Follow

edited Jan 29, 2009 at 19:35

answered Jan 29, 2009 at 3:50

OscarRyz

199k ● 119 ● 396 ● 573

1

If

```
public class BEvent extends AEvent {
...
}

public interface BListener extends AListener {

  public void event2(BEvent event);
}
```

can't you do somethin like:

```
public class B extends A {

  @Override
  public synchronized void addAListener(AListener l) {
    if (l instanceof BListener) {
       ...
    } else {
       super.addAListener(l);
    }
  }
  ...
}
```

As I've said in the comment, I'm not sure what do you actually want to achieve? Who is called from where, and what does it need to do when it get called?

Share Improve this answer Follow

answered Jan 27, 2009 at 14:43

Slartibartfast

8,815 ● 6 ● 43 ● 45

1

Based on what little information we have about the relationship between `A` & `B`, I think it's confusing to make `BListener` a subinterface of `AListener`. As the name suggests, a `BListener` is suppose to listen for `BEvent`s, which are *already* a subclass of `AEvent`s. For clarity, listeners should have discerning purposes; they shouldn't overlap unnecessarily. Besides, there is no need for such overlapping listeners since you already define separate methods in class `B` to handle different types of listeners.

To illustrate my point, consider this example, styled after your code:

```
public class MovableMouseEvent extends EventObject

public class ClickableMouseEvent extends MovableMouseEvent

public interface MovableMouseListener extends EventListener
  // mouseMoved(MovableMouseEvent)

public interface ClickableMouseListener extends MovableMouseListener
  // mouseClicked(ClickableMouseEvent)

public class MovableMouseWidget
  // {addMovableMouseListener,removeMovableMouseListener}(MovableMouseListener)
  // fireMovableMouseEvent(MovableMouseEvent)
```

```
public class ClickableMouseWidget extends MovableMouseWidget
  // {addClickableMouseListener,removeClickableMouseListener}
(ClickableMouseListener)
  // fireClickableMouseEvent(ClickableMouseEvent)
```

This design works, but is confusing because `ClickableMouseListener` handles two kinds of events, and `ClickableMouseWidget` handles two kinds of listeners, as you've rightly pointed out. Now, consider the following alternative which uses composition instead of inheritance:

```
public class MouseMoveEvent extends EventObject // note the name change

public class MouseClickEvent extends EventObject // don't extend MouseMoveEvent

public interface MouseMoveListener extends EventListener
  // mouseMoved(MouseMoveEvent)

public interface MouseClickListener extends EventListener // don't extend
MouseMoveListener
  // mouseClicked(MouseClickEvent)

public interface MouseMoveObserver
  // {addMouseMoveListener,removeMouseMoveListener}(MouseMoveListener)
  // fireMouseMoveEvent(MouseMoveEvent)

public interface MouseClickObserver
  // {addMouseClickListener,removeMouseClickListener}(MouseClickListener)
  // fireMouseClickEvent(MouseClickEvent)

public class MovableMouseWidget implements MouseMoveObserver

public class ClickableMouseWidget implements MouseMoveObserver,
MouseClickObserver
```

Share

Improve this answer

Follow

edited Jan 30, 2009 at 1:26

answered Jan 29, 2009 at 15:19

Zach Scrivena

29.5k ● 12 ● 65 ● 73