Delegates Reloaded: Walking the Path

Using the reflection API

by Michael Birken

The function pointer, a powerful concept in the C and C++ programming languages, has no direct equivalent in Java. No syntax exists to pass the address of a method to a JButton, for instance, that links it with pressing the button. Instead, Java promotes the use of anonymous inner classes, like this one:

```
okButton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    startProcessing();
  }
});
```

Back in October 1996, in an attempt to eliminate the need for this bloated syntax, Microsoft introduced an object-oriented method pointer into J++ called a "delegate." Sun Microsystems, citing the delegate as language pollution, sued Microsoft a year later for violating its Java license agreement. The lawsuit successfully maintained the purity of Java, but it also encouraged Microsoft to develop a competing Java-like language, C#. Today, the delegate lives on in C# and the other .NET programming languages that comprise Microsoft's Common Language Runtime.

Had the delegate become part of Java, would Swing programming be easier? More than simply hooking a component to an action, the delegate could invoke methods directly or asynchronously on a worker thread. Could that technique have solved many of the Swing threading headaches that we're faced with today? This article explores these possibilities by using Java's reflection API to restore the delegate concept to Java.

Building a Delegate

java.lang.reflect.Method provides a description of a method including formal parameter types, return type, access modifier, declared exceptions, and annotations. It also provides an invoke() method to perform an indirect call; however, unlike a function pointer, Method is not bound to a particular object instance. It's part of a toolkit that includes Constructor and Field for inspecting classes. To use invoke() you must pass in a reference to an object instance.

Listing 1 demonstrates how to use it to call System.out.println() indirectly. Since the static member variable out is an instance of PrintStream, Line 4 calls getMethod() on the PrintStream Class passing in a description of the method that we're interested in. Note, many of the reflection API methods were retrofitted with varargs as of Java 5; previously, getMethod() accepted the method name followed by the method parameter types as a Class[]. Line 8 passes the string to print along with out to invoke() on the Method acquired in line 4.

To build a Delegate class, I joined together a Method and an object instance. Compare Listing 1 to the following snippet:

```
Delegate delegate = new Delegate(
  System.out, "println");
delegate.invoke("Hello World!");
```

First, notice the lack of exception handling. Internally, Delegate transforms checked exceptions into unchecked exceptions for cleaner code. Second, observe that when Delegate is constructed, the println() method is not fully described; the parameter types are absent. Delegate.invoke() completes the method resolution on the first call using getClass() on each argument. Subsequent calls reuse a cached Method; however, this technique, although convenient, fails when a null parameter is used on the initial call. An alternative constructor lets you specify the argument class types explicitly like this:

```
Delegate delegate = new Delegate(
  System.out, "println", String.class);
delegate.invoke("Hello World!");
```

Delegate seeks out and binds to methods of any access level, including private ones. Instead of using Class.getMethod(), which obtains only public methods,

Michael Birken is actively involved in the design and research of emerging trading technologies at a Manhattan-based financial software company. He is a Sun Certified Java Programmer and Developer. He has a BS in computer engineering from Columbia University.

o_1@hotmail.com

Delegate calls Class.getDeclaredMethod(). Unfortunately, both of those functions exclude inherited methods. To get around that, Delegate iteratively applies getDeclaredMethod() to all the classes in the hierarchy with the aid of Class.getSuperclass(). After locating the method of interest, Method.setAccessible(true) is used to suppress invocation access checking.

If you're surprised that you can call a private method from outside of an object, recognize that access modifiers were designed for code organization, not for security; they enable you to expose usage method while hiding implementation details. Suppressing access checking is highly beneficial for unit testing. Normally to unit test implementation methods, you have to grant them at least default (package) access; however, Delegate provides an alternative that lets you keep those methods private.

Since static methods are bound to classes, not instances, Delegate provides another constructor, which accepts a Class. The following example creates a Delegate to the Math.cbrt() function, a static method introduced in Java 5 to compute cube roots.

```
Delegate delegate = new Delegate(Math.class, "cbrt");
double x = (Double)delegate.invoke(8.0);
```

Delegate.invoke() returns type Object; in this case, the result is first cast to Double and then auto-unboxed to a double primitive.

Bridging Tiers

Delegate's primary purpose is to serve as a bridge between an event-driven front-end and a multithreaded middle tier. In Swing, when an event is triggered, it does not execute right away. Instead, it joins the event queue (java.awt.EventQueue) where it waits along with other events to be executed by the Event-Dispatch Thread (EDT). The EDT pulls events off the queue and services them one-by-one.

An event that requires a significant time to complete – either because it's computationally intensive or because it makes blocking calls – holds up the rest of the queue, causing the user interface to seem sluggish. In fact, since repaint requests traverse the queue as well, expensive events typically cause components to appear as lifeless solid-colored rectangles.

If an event can't be processed in a timely fashion, it should forward the request to a worker thread. That's the reason why Delegate provides invokeAsync(). invokeAsync() is called just like invoke(), but it dispatches a java.util.concurrent.ThreadPoolExecutor thread to execute the method. Since it's an asynchronous call, it returns immediately without a return value. To see this in use, here's an indirect call to println() on a different thread:

```
Delegate delegate = new Delegate(
  System.out, "println");
delegate.invokeAsync("Hello World!");
```

Now, Swing components aren't thread-safe; they were designed to be accessed exclusively by the EDT. To safely call Swing methods from worker threads, Delegate provides invokeUI(). If invokeUI() is called by the EDT, the method is invoked directly. Otherwise, a request to execute the method is placed in the event queue. Since the call is potentially asynchronous, invokeUI() doesn't return a value. On the other hand, Delegate.invokeUIAndWait() enqueues the request, blocks until the EDT fulfills it, and then returns the result. The following example demonstrates how to obtain a JTextField value safely from any thread:

```
Delegate delegate = new Delegate(
  textField, "getText");
String text = (String)delegate
  .invokeUIAndWait();
```

Constructors

`Delegate(Object object, String methodName)`	Instance method Delegate. Resolution completes on initial call.
`Delegate(Object object, String methodName, Class... argumentTypes)`	Fully specified instance method Delegate.
`Delegate(Class _class, String methodName)`	Static method Delegate. Resolution completes on initial call.
`Delegate(Class _class, String methodName, Class... argumentTypes)`	Fully specified static method Delegate.

Methods

`Object invoke(Object... arguments)`	Call Delegate method directly on same thread.
`void invokeAsync(final Object... arguments)`	Executes Delegate method on worker thread.
`void invokeUI(final Object... arguments)`	Executes Delegate method on EDT.
`Object invokeUIAndWait(final Object... arguments)`	Executes Delegate method on EDT and waits for the result.

Figure 1 Constructors and methods of Delegate

A Listener Delegate

To join Swing events with methods, I combined delegates with a dynamic proxy. The dynamic proxy is an often-overlooked concept that's been available since Java 1.3. Before I discuss it, let's quickly review the proxy pattern.

A proxy acts as a middleman, serving as a stand-in for another object. Typically that other object and the proxy share a common interface. In the case of Remote Method Invocation (RMI), for example, you access a remote object living in a JVM on a different machine across the network via a local proxy. Internally, the local proxy forwards all method calls to the remote counterparts. The proxy effectively creates the illusion that the remote class is accessible locally.

For Swing components, I needed a proxy with the ability to stand-in for any event listener. I could have created a class that implements all of the listener interfaces in java.awt.event and forwards the calls to Delegate.invoke() accordingly, but that would have required way too much typing.

Luckily, java.lang.reflect.Proxy provides a static newProxyInstance() method for generating a proxy class on-the-fly that implements a specified set of interfaces – hence the term "dynamic proxy." newProxyInstance() requires a ClassLoader, a list of interfaces as a Class[], and a reference to an InvocationHandler implementation. InvocationHandler contains a single method with this signature:

```
public Object invoke(Object proxy,
  Method method, Object[] args)
    throws Throwable
```

All calls on the dynamically created proxy funnel down to InvocationHandler.invoke() where the Method parameter contains a description of the invoked proxy method. It's a simple matter to use it along with args to make a Delegate.invoke() call.

To generate dynamic proxies, I created a factory class called UIDelegate. Listing 2 shows one of its static create() methods. The return type, UIDelegateListener, is an interface that extends all of the interfaces in java.awt.event. UIDelegateProxy is an inner class that implements InvocationHandler.

The create() method accepts an arbitrary number of arguments in groups of four, each corresponding to a method of an event listener interface. A group consists of the object with a handler method, the name of the event method, the name of the handler method, and the technique to call the handler. The last parameter is a boolean: true indicates that the handler is invoked asynchronously and false causes the EDT to invoke it directly. The following example shows how to associate a handleOK() method with a button:

```
okButton.addActionListener(UIDelegate
  .create(this, "actionPerformed",
    "handleOK", true));
```

UIListener create(Object... params)	A group for each listener method: (1) object/class, (2) listener method, (3) handler method and (4) synchronous/asynchronous call flag.
UIListener create(Object object)	Object containing instance handler methods.
UIListener create(Object object, boolean asyncUIHandling)	Object containing instance handler methods and synchronously/asynchronous call flag.
UIListener create(Class _class)	Class containing static handler methods.
UIListener create(Class _class, boolean asyncUIHandling)	Class containing static handler methods and synchronous/asynchronous call flag.

Figure 2 Static factory methods of UIDelegate

Dialog Patterns

Switching from the EDT to a worker thread and back again to complete a business task maintains user interface responsiveness, but it breaks up a conceptually linear flow of execution into scattered, disjoint pieces. Consider a menu item that pops up in an options dialog before executing a task. To avoid the threading issues discussed in body of this article you can link the OK button on the dialog to the business logic with Delegate.invokeAsync(). When the task completes, it callbacks a success or failure method with Delegate.invokeUI(). That technique certainly works and often can't be avoided, but whenever possible, strive for a linear flow of execution.

One way to achieve this is to follow the pattern used by JOptionPane.showInputDialog(). showInputDialog() displays an input dialog and it blocks until the user enters a string. A complicated options dialog, as mentioned in the previous example, would return a bean containing a set of fields. We can link the menu item that manifests the dialog to a controller method using UIDelegate and we can configure it to call the controller with a worker thread. The controller provides the linear flow: it shows the dialog, it blocks for input, it validates the input, and it either displays an invalid fields message or it runs the business logic. The show() method of such a dialog must be thread-safe, unlike showInputDialog() which is designed to be exclusively invoked by the EDT. See the show() method of ProgressDialog for an example of how to do this.

The Mandelbrot Algorithm

The Mandelbrot fractal is the intersection of mathematics, computer science and art. It's amazing that something so visually attractive is generated by such a simple algorithm.

The fractal relies on basic properties of complex numbers. A complex number is a mathematical construct analogous to an object with two fields. It's written as the sum x + yi, where x and y are real numbers (double types) and i is the imaginary constant defined by the relation $i^2 = -1$. Each coordinate [x, y] represents a point on the complex plane. The magnitude of a complex number is the distance between that point and the origin, computable using the Pythagorean Theorem.

The image is generated by row scanning a rectangular region of the complex plane and assigning each point C = [x, y] a color by repeating the rule $Z_{N+1} = Z_N^2 + C$, where $Z_0 = C$ until the magnitude of Z_{N+1} exceeds 2. The value of N when the loop terminates determines the color of point C. If the loop fails to terminate after a larger number of iterations, then C is part of the Mandelbrot set and that point is traditionally assigned the color black. For those whose algebra is rusty, given C = [x, y] and Z_N = [a, b], $Z_{N+1} = Z_N^2 + C = (a + bi)^2 + (x + yi) = a^2 + 2abi + (bi)^2 + x + yi = a^2 + 2abi - b^2 + x + yi = [a^2 - b^2 + x, 2ab + y]$.

For further fractal exploration, see the references.

The referenced handleOK() method must accept an ActionEvent.

For those listeners with multiple methods, you only need to specify the methods that you're interested in. For example, MouseListener provides five methods, but if you only need to respond to enter and exit events on the EDT, you can do it like this (see Figure 2):

```
panel.addMouseListener(new UIDelegate(
  this, "mouseEntered", "handleEnter",
  false, this, "mouseExited",
  "handleExit", false));
```

Monitoring Progress

To demonstrate delegates in action, I put together a simple fractal image explorer. The application consists of an image window and a progress dialog. When you click on a point of the fractal, the progress dialog in Figure 3 appears, and a worker thread starts to compute a zoomed in region. The Cancel button lets the user terminate the running computation.

I opted not to use javax.swing.ProgressMonitor because it's not modal, it's difficult to get it to appear, and its cancel button causes it vanish immediately. Instead, I created a ProgressDialog class to encapsulate a dialog with a JLabel, a JprogressBar, and a JButton.

ProgressDialog monitors the progress of an object implementing IProgress, the interface in Listing 4. ProgressDialog contains a Swing timer. Every 0.25 seconds, it updates the progress bar using IProgress.getCurrent(). If the user presses Cancel, it invokes IProgress.requestCancel(), sending out a request that may not be satisfied immediately. The dialog remains visible until IProgress.isDone() return true, indicating that either the computation completed fully or it was terminated gracefully by the user.

Note: the implementation of IProgress must be thread-safe and its methods must return rapidly. For instance, in the fractal explorer, requestCancel() sets a volatile cancel-request flag instead of blocking until the worker thread terminates.

A cycle starts when a mouse click launches a worker thread using UIDelegate:

```
UIDelegate.create(this, "mouseClicked",
  "zoom", true)
```

Before computing the fractal, the worker thread calls the nonblocking ProgressDialog.show() method. show() safely manifests the dialog and kicks off the Swing timer by forwarding the call with Delegate.invokeUI(). The timer is linked to checkProgress(), shown in Listing 3, using:

```
UIDelegate.create(this,
  "actionPerformed", "checkProgress",
  false)
```

When isDone() return true, the optional ProgressDialog constructor parameter, completedDelegate, is called back on the EDT with the final progress value. If the completed progress is 100%, the image is obtained using IProgress.getResult(), and pasted to the window.

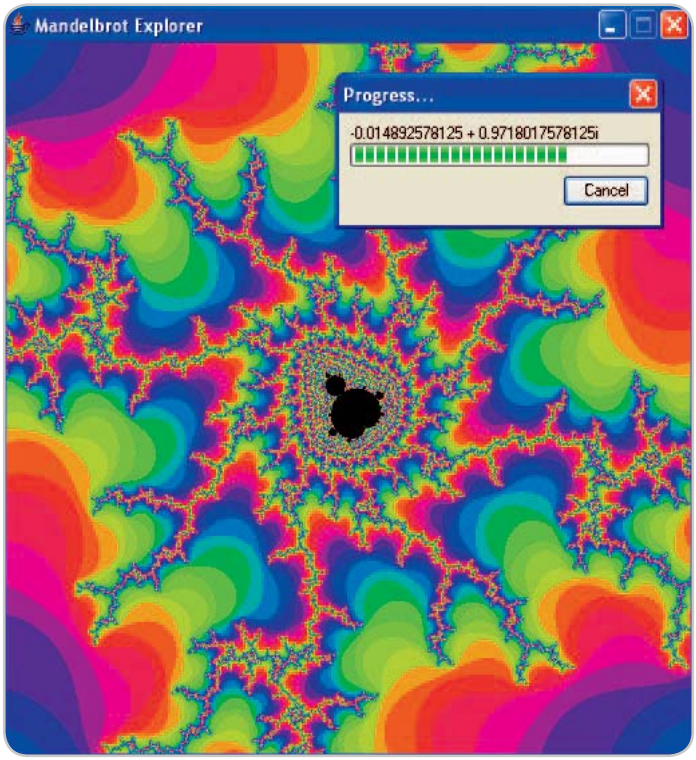

Figure 3 Modal dialog displays image generation progress

See the sidebar for details on the image-generation algorithm.

Pros and Cons

Delegates provide cleaner connections between components and handlers both syntactically and threadwise. The progress monitor example demonstrates how you can focus more on visual design and business logic than on juggling threads. However, Delegate is not type-safe; it's easy to get a runtime exception, for example, after misspelling a function name. Adding a type-safe delegate to a future version of Java might be asking for too much, but what about a method keyword, something like this:

```
Method m = this.actionPerformed(
    ActionEvent.class).method;
```

No need to catch a NoSuchMethodException since the check occurs at compile time.

Also, we're always told to avoid reflection because of its effect on performance. It's conceivable that searching for the Method corresponding to a specified name and a set of parameters takes a performance hit, but once located, what's the penalty of the Method.invoke() call itself? To gain a rough sense the answer, I used System.nanoTime() in a simple test harness like this:

```
long start = System.nanoTime();
for(int i = 0; i < NUM; i++) {
  // ... direct/indirect method call
}
long end = System.nanoTime();
double average = (end - start)
    / (double)NUM;
```

I ran my tests using Sun's Java 5 JVM with the default settings on a 1.8GHz PC running Windows XP. On my machine, it takes about 3.5ns (3.5 _10^{-9} seconds) for a normal method call and about 150ns for a delegated call. Although that's around 43 times slower, it still means that you can make 6.6 million delegated calls a second. Swing applications, at the very least, should be able take full advantage of Delegates without noticeable delays.

You can checkout my performance test harness along with the other code discussed in this article online at www.sys-con.com/java/sourcec.cfm.

References

- *.NET Delegates*: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/csref/html/vcrefTheDelegateType.asp
- *Reflection API Tutorial*: http://java.sun.com/docs/books/tutorial/reflect/
- *Using a Swing Worker Thread*: http://java.sun.com/products/jfc/tsc/articles/threads/threads2.html
- *Explore the Dynamic Proxy API*: http://java.sun.com/developer/technicalArticles/DataTypes/proxy/
- *Using Dynamic Proxies to Generate Event Listeners Dynamically*: http://java.sun.com/products/jfc/tsc/articles/generic-listener2/index.html
- *How to Use Progress Bars*: http://java.sun.com/docs/books/tutorial/uiswing/components/progress.html
- *RMI Tutorial*: http://java.sun.com/docs/books/tutorial/rmi/
- *Mandelbrot Set*: http://mathworld.wolfram.com/MandelbrotSet.html

Listing 1

```
1  Method m = null;
2  try {
3    Class c = PrintStream.class;
4    m = c.getMethod("println",
         String.class);
5  } catch(NoSuchMethodException e) {
6  }
7  try {
8    m.invoke(System.out,
         "Hello World!");
9  } catch(IllegalAccessException e) {
10 } catch(InvocationTargetException e){
11 }
```

Listing 2

```
1 public static UIListener create(
      Object... params) {
2   return (UIListener)Proxy
        .newProxyInstance(
3     object.getClass()
          .getClassLoader(),
4     new Class[] { UIListener.class },
5     new UIDelegateProxy(params));
6 }
```

Listing 3

```
1  private void checkProgress(
       ActionEvent e) {
2    if (progress.isDone()) {
3      dialog.dispose();
4      timer.stop();
5      if (completedDelegate != null) {
6        completedDelegate.invoke(
             progress.getCurrent());
7      }
8    } else {
9      noteLabel.setText(
         progress.getNote());
10     progressBar.setValue(
         progress.getCurrent());
11   }
12 }
```

Listing 4

```
 1 public interface IProgress {
 2   public void reset();
 3   public int getCurrent();
 4   public boolean isDone();
 5   public String getNote();
 6   public String getLargestNote();
 7   public void requestCancel();
 8   public int getMin();
 9   public int getMax();
10   public Object getResult();
11 }
```