Dynamic Execution Framework DEF

User's Guide

Dec 12, 2014

joe@javajoemorgan.com

Table of Contents

Introduction

I began developing this API many years ago when I was doing consulting work with Java. At the time, the only generally integrated asynchronous capabilities were built into Swing, the UI API. I needed asynchronous capabilities with every day programming, and wondered why the architects and programmers at Sun who were responsible for Java didn't think beyond the box and had not provided such a API.

This is the reason why this API came into existence. The aim was to create a simple, but powerful, asynchronous and dynamic set of utilities useful both within the UI and non-UI environments. Another goal was to remove the bloat and redundancy of having to create and start threads, and then figuring out how and where to write a singleton `run` method when I just simply needed to be able to asynchronously execute several methods of a third party API.

Sometime later, Sun released the Concurrency API, but by that time, I was well on my way to what I continue to consider a much better asynchronous and dynamic set of tools. Though initially excited, I soon realized that the Concurrency API continues the same process of relying very heavily on implementations not belonging to the programming task at hand. This new API requires the creation of bloated and semantically meaningless implementations of `Runnable` and `Callable` to get the job done. Other problems, such as the inability to restart a completed task, such as in the case of a `FutureTask`, is also a problem with the Concurrency API.

So, I stopped converting to the Concurrency API, which is not something I generally do, by the way, and resumed development of this set of utilities. I came to name this framework, the Dynamic Execution Framework, or DEF. With this API, you will be able to again just write your app the way an app should be written, without concerns about how you may later need to invoke methods asynchronously. With these utilities, there is no need to implement interfaces or do anything out of the ordinary. It allows you to simply focus on the goals of the application and the classes it contains and needs.

What is the lesson we are all taught within Object Orientation? “A class is something that does one thing and does one thing well.” Meeting the goals of this statement is not so easy when you start to concern yourself over how methods of a class may need to be invoked asynchronously, or if methods of a class need to invoke things dynamically. Do not read this to mean you should not write Java classes and beans that are properly synchronized, you should, but that is the easy part. When using the DEF, in fact, that's all you need to do!

There are many advantages to using the DEF over the facilities within the Concurrency API. First, you never again will need to subclass `Thread`. Second, you never again will need to implement `Runnable`. Third, you will soon be able to remove all out-of-place and off-topic “wrappers” and other code bloat and simply invoke the methods you wish to invoke in a much more direct and meaningful way. Asynchronous and dynamic invocations can occur at will, all without the need of a `ThreadFactory`, or having to develop `Runnable` or `Callable` “wrappers” around methods that were not designed to be run asynchronously, but may need to be.

Another problem with the threading model, as I see it, is having to wait on other threads, not knowing if, when, they will ever complete, and having to establish locks, or write strange loops or implement other "tricks" to get things working correctly. Once you understand the idea of the `InvocationListener`, and you see the much superior `Timer` with `TimerEvent`, and when you create your first Java function pointer, you will be ecstatic!

In most cases, it takes a single line of code to asynchronously execute a method. Any method. Even methods of third party classes. Even non-public methods (but respects rules when there are SecurityManager restrictions). Even methods that take arguments! Even methods that take arguments whose values should not be derived until the moment the method is called, even if the execution of the method is delayed for an indeterminate period of time! This is also true of static methods.

For a much clearer demonstration for this use case, the examples in the Concurrency API documentation of the `Future` interface require at least 12 lines of code just to get a search to run asynchronously. Furthermore, it requires 3 class/interface references; one to `ExecutorService`, another to `Future`, and an anonymous bloated implementation of `Callable`. (Let's not get into how bad of an idea I think anonymous classes are. In my opinion, they should be banned in exactly the same way `goto` is.)

In any case, rather than:

```
interface ArchiveSearcher { String search(String target); }
 class App {
   ExecutorService executor = ...
   ArchiveSearcher searcher = ...
   void showSearch(final String target)
       throws InterruptedException {
     Future<String> future
       = executor.submit(new Callable<String>() {
         public String call() {
             return searcher.search(target);
         }});
     displayOtherThings(); // do other things while searching
     try {
       displayText(future.get()); // use future
     } catch (ExecutionException ex) { cleanup(); return; }
   }
 }
```

With this API, you only need:

```
AsynchMethodInvoker mi = new AsynchMethodInvoker(searcher, "search", target);
displayOtherThings();
if (mi.hasCompleted()) displayText(mi.getReturnValue());
```

If you like the DEF in this context, you will love it within a UI, such as within a Swing application. The API includes a set of classes categorized as "`EventInvokers`", and with them, you may never again need to implement `ActionListener`, `PropertyChangeListener`, `MouseListener`, `KeyListener`, `ComponentListener`, or any of the other commonly used event listening interfaces. With usually a single line of code, you can direct the flow of execution to the exact place you want it when the event occurs. No more interfaces! No more stub implementations! These are true event delegates enabling you to create cleaner code.

I hope you enjoy using these tools. I have saved thousands of lines of code with these simple classes, and I am certain you will thoroughly enjoy them too.

Please write with questions, suggestions, enhancement requests, bugs, typographical errors and other concerns to joe@javajoemorgan.com. Source code is available for a price. Please write for pricing.

Chapter 1 – Get going now!

We'll start with the most commonly used classes and utilities and simple program examples of what they do. Later in this manual, we will cover things in more detail. If you are anything like me, you want to get going, rather than muddle through a bunch a techno gunk you can worry about later.

Primary Use Cases

There are two primary classes within this API: `MethodInvoker` and `AsynchMethodInvoker`. These two classes form the basis of virtually everything else within this API. They do pretty much as they are named, that is, they invoke methods. `MethodInvoker` gives you dynamic execution capabilities of a target method. `AsynchMethodInvoker` extends this class to provide asynchronous invocation capabilities. Both classes are essentially function pointers in Java, except that `AsynchMethodInvoker` is designed to go ahead to kick off the target method asynchronously, either right now or after some specified delay, whereas `MethodInvoker` invokes the method synchronously.

Let's get right to a program to show an example.

Using AsynchMethodInvoker

```
import com.jmorgan.lang.AsynchMethodInvoker;

public class SimpleAsynchExample {
   public static void main(String[] args) {
      // Invoke println later and print Hello World
      new AsynchMethodInvoker(System.out, "println", "Hello World");
      System.out.println("Normal Println");
   }
}
```

The above program asynchronously invokes the `println` method of the System's output stream passing in the string `"Hello World"` when invoked. To emphasize that the method is invoked in a separate thread, we have a normal `println` call. With a small change, we ensure the `println` method isn't invoked until 500 milliseconds later.

```
import com.jmorgan.lang.AsynchMethodInvoker;

public class SimpleAsynchExample {
   public static void main(String[] args) {
      // Invoke println 500 ms from now and print Hello World
      new AsynchMethodInvoker(System.out, "println", "Hello World", 500);
      System.out.println("Normal Println");
   }
}
```

So, as you can see, delaying an asynchronous invocation is just a few characters away from just invoking asynchronously. In both cases, the target method is invoked using a dedicated thread, but one is invoked at a later time.

Let's review the above code examples in a bit more detail, most specifically what the code does not contain. Notice there is only one class reference, no interface implementations, no wrappers, no subclasses of `Thread`, and no anonymous implementations! Just simple and direct code. In both cases, the first parameter to `AsynchMethodInvoker` is a reference to an object. The second parameter is a textual name of the method to invoke, in this case, the `println` method. The third parameter will be passed into the `println` method. In the second example, the fourth parameter represents an amount of time to wait, in milliseconds, before the method will be invoked.

Using MethodInvoker

`AsynchMethodInvoker's` polite brother is `MethodInvoker`. `MethodInvoker` is very similar to use, but instead of it automatically running the method in another thread, it requires you to explicitly run it via the `invoke` method:

```
import com.jmorgan.lang.MethodInvoker;

public class SimpleDynamicExample {
      public static void main(String[] args) {
            MethodInvoker mi =
                  new MethodInvoker(System.out, "println", "Hello World");
            mi.invoke();
      }
}
```

The mechanism for using `MethodInvoker` is virtually identical to `AsynchMethodInvoker` except for the need to explicitly tell the `MethodInvoker` instance to invoke it's method. There is no delayed timing parameter for `MethodInvoker`, and `MethodInvoker` doesn't invoke the target method in its own thread. `MethodInvoker` is somewhat like a function pointer, and we will show how to use it more effectively in later examples.

The above example doesn't give the DEF much justice, because you are thinking that you can just write:

```
      System.out.println("Hello World");
```

But, what if you want to invoke it asynchronously? You could do as so many do and write something like this:

```
     new Thread(new Runnable() {
       public void run() {
            System.out.println("Hello World");
       }
     }).start();
```

The point is, when you do need to dynamically invoke a method, or you need something tantamount to a function pointer, `MethodInvoker` becomes extremely useful. In day to day use, I use `AsynchMethodInvoker` way more than `MethodInvoker`, but you will come to see the benefits of having both.

Using ThreadUtility

I am making a quick reference to another class included within the DEF that I use frequently, but most of you likely have a similar utility or method. This is the `ThreadUtility` class, which you will see in the code samples. Basically, instead of:

```
try { Thread.sleep(1000); }
catch (InterruptedException e) { }
```

You will see:

```
ThreadUtility.sleep(1000);
```

Which is the same thing, but without the need for the `try/catch`. Saves a little bit of typing, but if you write applications where you frequently sleep, a little bit of typing adds up quickly.

That isn't all `ThreadUtility` can do, though. Other methods of the `ThreadUtility` class are:

```
public static int getThreadCount();
```

Which returns the total number of threads the system has, at the moment. Be careful on this one, as chances are better than not that this number has already changed by the time your program can consume it. It is, nonetheless, a useful number for general monitoring.

```
public static int getThreadCount(ThreadGroup threadGroup);
```

This returns the total number of threads for the given `ThreadGroup`. Again, this count will more than likely have changed before you can do much with the number.

```
public static void killThreads(String name);
```

As it sounds, the `killThreads` method kills all threads having a given `name`. This brings me to a related point, too. When `AsynchMethodInvoker` creates the thread for invocation, the thread's name is constructed from the name of the class of the target object combined with the name of the method being invoked. Therefore, the name of the thread created for the invocation of this:

```
new AsynchMethodInvoker(System.out, "println", "Hello World");
```

Is:

```
java.io.PrintStream.println
```

I added this sometime back when I wrote my own internet search engine as a proof of concept for the DEF. There is a little UI where you enter the starting page and a regular expression, then the thing goes off searching. While doing so, it creates hundreds of threads, some are searching, some parsing pages looking for links, others are reading pages, etc. I needed a way of stopping the process, which meant killing anything running under specific names. It works like a charm! Maybe you can find a use for it as well.

Handling Multiple Parameter Passing

Both `AsynchMethodInvoker` and `MethodInvoker` can pass parameters into their target method, but the mechanism of doing so is a little bit different, mainly because `AsynchMethodInvoker` has an optional delay:

```
import java.util.Calendar;
import com.jmorgan.lang.AsynchMethodInvoker;
import com.jmorgan.lang.MethodInvoker;

public class MultipleParamDynamicExample {
   public static void main(String[] args) {
      Calendar c = Calendar.getInstance();

      // Dynamically invoke one of the "set" methods of Calendar
      MethodInvoker mi = new MethodInvoker(c, "set", 1998, Calendar.MARCH, 25);
      mi.invoke();

      System.out.println("The Date After Dynamic Invocation Is: " + c.getTime());

      // Dynamically and asynchronously invoke the same method, different date
      AsynchMethodInvoker ami =
         new AsynchMethodInvoker(c, "set",
              new Object[] { 2000, Calendar.DECEMBER, 7 });

      while (!ami.hasCompleted()) ThreadUtility.sleep(100);

      System.out.println("The Date After Asynchronous Invocation Is: " +
                          c.getTime());
   }
}
```

Let's first review the differences, and then the reason why. Note that `MethodInvoker` takes any number of parameters following the name of the method. The signature of this constructor for `MethodInvoker` is:

```
public MethodInvoker(Object object, String methodName, Object...arguments)
```

All of the parameters following the name of the target method will be passed into that method, provided there is a method of that name and it receives the number and types of parameters passed. In the above example we are invoking the `set(int year, int month, int date)` method of the `Calendar` class. With `MethodInvoker`, we can just pass them in.

`AsynchMethodInvoker`, on the other hand, takes an optional delay, and so multiple parameters must be handled differently, otherwise, it is impossible to know the difference between an integer intended as a parameter to the target method and an integer intended as the delay. The signature of the constructor used in the above example is:

```
public AsynchMethodInvoker(Object object, String methodName, Object[] arguments)
```

But the constructor to do the same thing with a delay is:

```
public AsynchMethodInvoker(Object object, String methodName, Object[] arguments,
                           int delay)
```

So, as you can see, if we had wanted to invoke the `set` method with a one second delay, it would have looked like this, with the final integer indicating the delay, in milliseconds:

```
AsynchMethodInvoker ami = new AsynchMethodInvoker(c, "set",
                              new Object[] { 2000, Calendar.DECEMBER, 7 },
                              1000);
```

Obtaining Return Values

You can obtain the return value of the methods being invoked with both `MethodInvoker` and `AsynchMethodInvoker`. `MethodInvoker`'s invoke method returns the value returned by the target object's target method.

Since `AsynchMethodInvoker` runs the method in a separate thread, we need a different way of getting to the return value. First, we will use the `hasCompleted` method to determine if the method has been invoked, and then we can use the `getReturnValue` method to get the actual value. Here is a code example:

```
import java.util.Calendar;

import com.jmorgan.lang.AsynchMethodInvoker;
import com.jmorgan.lang.MethodInvoker;
import com.jmorgan.util.ThreadUtility;

public class GettingReturnValuesExample {
   public static void main(String[] args) {
      Calendar c = Calendar.getInstance();
      c.set(1982, Calendar.MAY, 15);

      MethodInvoker<Integer> mi =
             new MethodInvoker<Integer>(c, "get", Calendar.DATE);
      int date = mi.invoke();

      System.out.println("The date is: " + date);

      AsynchMethodInvoker<Integer> ami =
             new AsynchMethodInvoker<Integer>(c, "get", Calendar.MONTH, 0);

      while (!ami.hasCompleted()) ThreadUtility.sleep(100);

      int month = ami.getReturnValue();

      System.out.println("The month is: " + month);
   }
}
```

One very important and interesting note on the example for `AsynchMethodInvoker`. When you need to pass a single parameter, and that parameter is an `int`, you must also provide a delay, even if it is 0, as you see in the example. This is so the run-time can distinguish between these two constructors:

```
public AsynchMethodInvoker(Object object, String methodName, int delay);
public AsynchMethodInvoker(Object object, String methodName, Object argument);
```

As you can see, if we were to have just written:

```
AsynchMethodInvoker<Integer> ami =
     new AsynchMethodInvoker<Integer>(c, "get", Calendar.MONTH);
```

Since `Calendar.MONTH` is an `int` constant, the first constructor of the two constructors listed above is being used, meaning the `AsynchMethodInvoker` is looking for a `get` method of Calendar that takes no parameters, and it will fail. So, in order to pass a single `int` parameter into a target method with `AsynchMethodInvoker`, you must supply a delay value, even if zero. No other parameter type is affected.

Dealing with Methods Receiving Variable Argument Lists

The same kind of problem can exist when invoking methods that take a variable number of parameters. Consider an invocation of the `printf` method of `PrintStream`. The signature of `PrintStream's printf` method is this:

```
public PrintStream printf(String format, Object ... args)
```

Which technically, is two parameters; a String and an array of objects. In order to properly handle this, you'll need to bundle the list of parameters into an array for it to work properly:

```
import com.jmorgan.lang.AsynchMethodInvoker;
import com.jmorgan.lang.MethodInvoker;

public class VariableArgsExample {
   public static void main(String[] args) {
      // Dynamic invocation of
      //
      // System.out.printf("Hello %s, I have %d dollars!", "Joe", 10);
      MethodInvoker mi = new MethodInvoker(System.out, "printf",
                 "Hello %s, I have %d dollars!\n",
                 new Object[] { "Joe", 10 });
      mi.invoke();

      // Asynchronous invocation of the same
      new AsynchMethodInvoker(System.out, "printf",
                 new Object[] { "Hi %s, I have %d dollars!\n",
                 new Object[] { "Mary", 20 }});
   }
}
```

It is easier and clearer what to do within `MethodInvoker`. The first parameter following the target method name within the use case for `MethodInvoker` is the format, and the second is the array of objects used to fill in the pattern. For `AsynchMethodInvoker`, though, because multiple arguments need to be designated as an array of objects, and in this case since one of the parameters is itself an array of objects, the special case of an array containing an array is necessary.

But don't let these complications get you down. These are some of the most extreme cases of difficulty in using the DEF, but I did want to be assured you had clear examples and explanations on how to handle them.

Function Pointers

`MethodInvoker` and `AsynchMethodInvoker` are essentially function pointers in Java. I didn't invent this, the great folks at Sun did, but these classes hide some of the details of making dynamic invocations easier. Both classes hold a reference to an object, employ an algorithm to resolve the method to invoke, and then invoke that method based upon the contract of the class. `MethodInvoker` invokes a method synchronously when its `invoke` method is called. `AsynchMethodInvoker` invokes a method in a separate thread when the class is instantiated.

Both instances can feed parameters to the target method, and both have their own nuances for handling cases when multiple parameters and variable argument lists need to be provided to the target method.

In the examples seen in Chapter 1, the values of the arguments provided to the target methods were established at the time they were provided to the methods. When used this way, this makes the parameter values somewhat constant. In many cases, this is just fine. In other scenarios, however, one needs the values of the arguments to the target methods to represent a more real time value.

Take, for example, a bean state monitor that, from time to time displays the values of a bean every second. This pattern has been very useful for debugging when a problem is temporal in nature, rather than simply procedural. However, this pattern is also necessary for UI's requiring regular visual updates based upon the state of one or more objects.

Here is a very simple example of the problem:

```java
import java.util.Calendar;

import com.jmorgan.lang.MethodInvoker;

public class ConstantParmValueProblem {
   public static void main(String[] args) {
      Calendar c = Calendar.getInstance();
      c.set(2011, Calendar.JANUARY, 15);

      MethodInvoker mi = new MethodInvoker(System.out, "println",
                                           c.getTime());
      mi.invoke();

      ThreadUtility.sleep(1000);

      c.set(1995, Calendar.AUGUST, 31);

      mi.invoke();
   }
}
```

Notice the date printed is January 15, 2011 in both cases, which may not be desirable or apparently intended by the path of the code. This is because the value of the parameter passed into the invocation of the `println` method is established when the `MethodInvoker` is instantiated. So `println` is printing the same value both times, even though the state of the original object has changed.

What we really want is the value *when* the method is invoked, every time and whenever it occurs. To solve this, we will show how `MethodInvoker` can be used as a surrogate for the parameter of the target method. More accurately, `MethodInvoker` will be used to dynamically invoke the `getTime` method of `Calendar` instance, and use the return value of that method as the parameter for the `println` method of the `System` output stream.

```
import java.util.Calendar;

import com.jmorgan.lang.MethodInvoker;

public class ConstantParmValueSolution {
  public static void main(String[] args) {
    Calendar c = Calendar.getInstance();
    c.set(2011, Calendar.JANUARY, 15);

    MethodInvoker getTimePointer = new MethodInvoker(c, "getTime");

    MethodInvoker mi = new MethodInvoker(System.out, "println",
                                         getTimePointer);
    mi.invoke();

    ThreadUtility.sleep(1000);

    c.set(1995, Calendar.AUGUST, 31);

    mi.invoke();
  }
}
```

As you can see, the correct date is printed. This is because the `getTime` method of the `Calendar` instance is accessed by a “pointer” to the method, rather than by capturing the return value of the method during the instantiation of `MethodInvoker`, which effectively makes the value of the parameter to the method a constant. The difference is created because when the `MethodInvoker` runs `println`, it needs to provide the method an argument. When it looks up the argument to provide to the `println` method, it sees that it is referencing another `MethodInvoker` instance. This instance is targeted at the `getTime` method of the `Calendar` instance. When `invoke` is called for the `println` method, it then runs `getTime` of the `Calendar` instance, retrieves the return value and then passes that value to `println`.

Finally, to bring this point together, we'll see how `MethodInvoker` and `AsynchMethodInvoker` can work together to produce a really nice asynchronous and dynamic result:

```java
import java.util.Calendar;

import com.jmorgan.lang.AsynchMethodInvoker;
import com.jmorgan.lang.MethodInvoker;

public class AsynchDynamicParmValueExample {
  public static void main(String[] args) {
    Calendar c = Calendar.getInstance();
    c.set(2011, Calendar.JANUARY, 15);

    MethodInvoker getTimePointer = new MethodInvoker(c, "getTime");

    new AsynchMethodInvoker(System.out, "println", getTimePointer);
    new AsynchMethodInvoker(c, "set",
                            new Object[] { 1995, Calendar.AUGUST, 31 }, 500);
    new AsynchMethodInvoker(System.out, "println", getTimePointer, 1000);
  }
}
```

In the above example, we initialize the date to January 15th, 2011. Then we setup a pointer to the `getTime` method of the `Calendar` instance using `MethodInvoker`, and then invoke `println` asynchronously. In order not to create a timing issue, and since we don't know exactly when it is safe to change the date, we invoke the `set` method 500 milliseconds later. Then again, to display its new value, we print the calendar's value one second after that.

Summary

OK, maybe that wasn't a perfect example, but it proves the application of the utility of this API in the simplest of ways. Keep in mind that `MethodInvoker` runs its target method synchronously. As a stand-alone class, it proves its worth mostly as a function pointer and surrogate for an asynchronous process, and is also very useful in environments where you know a method exists on an instance, but the reference you have to it is too primitive to resolve at compile time.

There have also been many times I have a reference to instances of classes that do not and can not exist within the same hierarchy or, for whatever reason, cannot implement the same interface because the classes are third-party classes where I don't have access to the source code. In these situations, being able to dynamically invoke one or more methods is priceless.

I'm going to run off and write Chapter 2 to tell you a bit more about the remaining features of `MethodInvoker` and `AsynchMethodInvoker`. While you're waiting, start browsing through your code for places where you can replace code bloat with what you've learned so far. I'll bet you can start shaving off hundreds of lines of code, thus reducing the complexity of your apps, thereby improving maintainability as well!

Chapter 2 – Additional Features of the DEF

No dynamic and asynchronous process would be complete without being able to cancel a scheduled process. Once the ability to cancel a process is introduced, especially in a multi-threaded world, the ability to know if the process has canceled is needed. You'll also really like the ability to restart an asynchronous process. We'll also take a look at the `InvocationListener` interface which gives us the ability to not have to loop and wait for the finalization of an asynchronous invocation, but simply be notified when it occurs. Finally, we will see it all come together with `DynamicProcess`.

Canceling an Asynchronous Process

So, you kicked off the process, then something happens that means it may not be such a good idea for it to actually run. You need a way to cancel the process if it has not yet run. Canceling is only possible if the invoking process has not actually invoked the target method, which means it only applies to the `AsynchMethodInvoker`. Additionally, it is possible to know if the process is canceled. Here is an example:

```java
import com.jmorgan.lang.AsynchMethodInvoker;
import com.jmorgan.util.ThreadUtility;

public class CancelProcessExample {
   public static void main(String[] args) {
      // Setup to run println 5 seconds from now
      AsynchMethodInvoker mi = new AsynchMethodInvoker(System.out, "println",
                                                "Hey, I wasn't canceled", 5000);

      System.out.println("Other things happening here....");

      // Tell it to cancel
      mi.cancel();

      // Waiting for final cancel notification may take a while
      while (!mi.wasCanceled()) {
         System.out.println("Hasn't even tried yet");
         ThreadUtility.sleep(1000);
      }

      // It will loop indefinitely above if cancel doesn't work,
      // so if you see this, everything worked just fine!
      System.out.println("Cancel Successful");
   }
}
```

Your results may vary, but generally speaking, if the program prints, `"Hey I wasn't canceled"`, then it didn't work. I venture to say, though, that not only did it work, but it probably didn't get a chance to print out that it `"Hasn't even tried yet"`.

Restarting an Asynchronous Process

It's finished! Done! Over! What do you mean it got canceled!? No, this isn't a rejection letter from a dream date. We're still talking about the DEF. For some reason, the Concurrency API and the mentality of the multi-threading universe sees this as the end of the world. Well, with the DEF, it isn't. Of course, recreating an asynchronous process is just as easy as initially creating it. It's just one line of code, after all.

Before you go off taking the short road and instantiating new `AsynchMethodInvoker`s all over the place, take a moment to understand a little more detail about what it is doing. When you write something like:

```
new AsynchMethodInvoker(System.out, "println", "Hello World");
```

You are creating an instance of an `AsynchMethodInvoker`, quite obviously, but what happens inside the guts of the thing? Well, it first determines if there is a method with the target name. Then, if arguments are provided, it must match up the arguments to the method by type. Anyone having the slightest bit of Reflection API experience knows how this is done. This API takes parameter to method matching quite a bit further, and as you use the API, you will come to understand just how far it goes.

Once it finds the matching method, it then proceeds to make any data conversions it deems necessary before invoking the method. Another nice thing the DEF does is it creates a cache of methods of classes by parameter. This way, if you do decide to create a series of instances rather than reuse the existing instance, future matching should occur faster. In short, it just saves you a bunch of code and worry.

Make no bones about it, though. Even though the DEF streamlines performance, any API like this will take some toll on speed. It will speed up your programming and maintenance, but not your program. Yet, when you need a friendly, fast, and simple approach to dynamic and asynchronous processes, this API is sufficient for the vast majority of applications.

Returning back to the topic at hand, once an `AsynchMethodInvoker` finishes, the thread used to invoke the method is, indeed dead. What do you care? If you want to invoke it again with the same settings as before, just do it! Here's how:

```
import java.util.Calendar;

import com.jmorgan.lang.AsynchMethodInvoker;
import com.jmorgan.util.ThreadUtility;

public class RestartingAsynchMethodExample {
  public static void main(String[] args) {
    Calendar c = Calendar.getInstance();
    c.set(1994, Calendar.APRIL, 1);

    AsynchMethodInvoker mi =
      new AsynchMethodInvoker(System.out, "println", c.getTime());

    while (!mi.hasCompleted()) ThreadUtility.sleep(100);

    // Just for fun, Let's delay the next execution for 1/2 second
    mi.setDelay(500);
    mi.invokeMethod();
  }
}
```

Using InvocationListener

Regardless of which flavor of invoker you are using, sometimes knowing when things are about to happen, or when they have happened is important. You've already seen a couple of ways to poll for information, such as the use of the `hasCompleted` and `wasCanceled` methods.

Another way to be able to react to the internals of what is going on within the DEF is to install one or more `InvocationListener`s. An `InvocationListener` is an interface whose implementations receive notificationa of `InvocationEvent`s. The invocation event contains all the information about the invocation, such as the target object and method, the values of the arguments to the method, the time of invocation, if the method returns a value, and the return value.

`InvocationListener`s have three events:

```
public boolean methodInvocationNotice(InvocationEvent invocationEvent);
public void methodInvoked(InvocationEvent invocationEvent);
public void methodInvocationCancelled(InvocationEvent incocationEvent);
```

The first event is fired just before the target method is invoked. Within the `InvocationEvent`, the time of the event is still `null` and there is no return value. If the `methodInvocationNotice` returns `true`, it indicates it is OK to invoke the target method. If the `methodInvocationNotice` returns `false`, the listener indicates the method should not be invoked. If multiple listeners are registered, and any one of their `methodInvocationNotice` events returns false, the method will not be invoked. This, then, is another way to cancel the method invocation.

The second event is fired after the target method on the target object is invoked and has returned. If the target method returns a value, that value is available to the `InvocationEvent`. The third event is fired only if the method invocation is canceled. The `InvocationEvent` indicates the invocation is canceled as well. Here is an example:

```
import com.jmorgan.lang.AsynchMethodInvoker;
import com.jmorgan.lang.InvocationEvent;
import com.jmorgan.lang.InvocationListener;
import com.jmorgan.lang.MethodInvoker;
import com.jmorgan.util.ThreadUtility;

public class InvocationEventExample  {
   public static void main(String[] args) {
      Listener listener = new Listener("Normal");

      MethodInvoker<Void> mi =
         new MethodInvoker<>(System.out, "println", "Hello There");
      mi.addInvocationListener(listener);
      mi.invoke();

      listener = new Listener("Cancelled");
      AsynchMethodInvoker<Void> ami =
         new AsynchMethodInvoker<>(System.out, "println",
                                    "I should be cancelled.", 250);
      ami.addInvocationListener(listener);
      // This to simulate some delay between the AsynchMethodInvoker
      //and its cancellation, otherwise the thread may not have been
      //invoked by the thread manager, and cancel would appear to fail
      ThreadUtility.sleep(50);
      ami.cancel();

      listener = new Listener("Revoked", false);
      ami = new AsynchMethodInvoker<>(System.out, "println",
                                       "I should be revoked.", 250);
      ami.addInvocationListener(listener);
   }

   static class Listener implements InvocationListener {
      private boolean shouldInvoke;
      private String eventType;

      public Listener(String type) {
         this(type, true);
      }

      public Listener(String type, boolean shouldInvoke) {
         super();
         this.eventType = type;
         this.shouldInvoke = shouldInvoke;
      }

      public void methodInvoked(InvocationEvent invocationEvent) {
         System.out.println(this.eventType + ": " + invocationEvent);
      }

      public boolean methodInvocationNotice(InvocationEvent invocationEvent) {
         System.out.println(this.eventType + ": " + invocationEvent);
         return this.shouldInvoke;
      }

      public void methodInvocationCancelled(InvocationEvent invocationEvent) {
         System.out.println(this.eventType + ": " + invocationEvent);
      }

   }
}
```

Note in the above example, the `Listener` class implements `InvocationListener`, and so provides the requisite implementation. Just before the method is to be invoked, the `MethodInvoker` notifies all `InvocationListener`s via `methodInvocationNotice`. When the `invoke` method is called with the

MethodInvoker, it notifies all registered InvocationListeners via methodInvoked handing them an InvocationEvent, which contains all of the information needed related to the invocation.

Important methods for the InvocationEvent class are:

```
public Object getInvocationTarget();
public Method getMethod();
public Object[] getMethodArguments();
public boolean hasReturnValue();
public Class<?> getReturnValueType();
public Object getReturnValue();
public boolean isCancelled();
```

The first three are needed if the same InvocationListener is listening to multiple invokers. The next three are needed if the listener is interested in doing something with the return value. If hasReturnValue returns false, then the method is declared as a void method and getReturnValue returns null. Otherwise, getReturnValue returns the value returned by the invocation of the method.

Note that if getReturnValue returns null, and hasReturnValue is true, then the method returned null. As previously mentioned, getReturnValue returns null if hasReturnValue is false, which means that the method does not return a value. Therefore, it is necessary to use hasReturnValue in conjunction with getReturnValue to distinguish the two cases.

isCancelled returns true if the method invocation was canceled either by external code, or by any registered InvocationListener.

Creating a Dynamic Process

So far, we have seen many uses of MethodInvoker and AsynchMethodInvoker. These are great when you have a single function to call or for use as a single function pointer. Not too long ago, however, I was working on some coding related to a wholesale business, and the application of taxes began to look somewhat like the tax code. For some, no tax was charged. For others, tax was charged on certain items but not others. Sometimes tax was charged before other times. It became, well, irritating.

So, I had a few choices. I could architect an interface/class hierarchy based upon dependency injection that would work, performing maybe a "computeTotal" function, and then create some kind of factory class to dish out the correct instance based upon the conditions of the transaction. I could have just written about a dozen or so functions and some kind of switch statement that called the correct function. Of course, either of those would work.

But then I thought, what if I could just dynamically build a list of pointers to the functions I needed to call, and arrange them in the correct order based upon the logic of the rules. So, I put together this little gem called a DynamicProcess. Here is a really simple example of how it works:

```
import com.jmorgan.lang.AsynchMethodInvoker;
import com.jmorgan.lang.DynamicProcess;
import com.jmorgan.lang.MethodInvoker;

public class DynamicProcessExample {
  public static void main(String[] args) {
     DynamicProcess dynamicProcess = new DynamicProcess();

     MethodInvoker mi1 = new MethodInvoker(System.out, "println",
                                           "First Message");
     MethodInvoker mi2 = new MethodInvoker(System.out, "println",
                                           "Second Message");
     MethodInvoker mi3 = new MethodInvoker(System.out, "println",
                                           "Third Message");

     dynamicProcess.addProcessMethod(mi1);
     dynamicProcess.addProcessMethod(mi2);
     dynamicProcess.addProcessMethod(mi3);

     dynamicProcess.invoke();

     new AsynchMethodInvoker(dynamicProcess, "invoke", 1000);
  }
}
```

Over-simplistic, I know, but shows us that we can assemble `MethodInvoker`s together dynamically and then invoke the bunch. In the last statement of the example, I show that we can mix-and-match to asynchronously invoke the dynamic process as well.

This can become really complex when you begin to envision that you can use `MethodInvoker`s as surrogates for parameters into the methods contained by the `DynamicProcess`! This way, if one or more of the methods within the `DynamicProcess` require parameters that need real-time values when they are invoked, you can put together a neat little package and not lose a bit of capability.

`DynamicProcess` comes with two constructors, the no-argument one you see above, and this one:

```
public DynamicProcess(MethodInvoker...methods)
```

So, the above example could have been written like this as well:

```java
import com.jmorgan.lang.AsynchMethodInvoker;
import com.jmorgan.lang.DynamicProcess;
import com.jmorgan.lang.MethodInvoker;

public class DynamicProcessExample {
  public static void main(String[] args) {
    MethodInvoker mi1 = new MethodInvoker(System.out, "println",
                                          "First Message");
    MethodInvoker mi2 = new MethodInvoker(System.out, "println",
                                          "Second Message");
    MethodInvoker mi3 = new MethodInvoker(System.out, "println",
                                          "Third Message");

    DynamicProcess dynamicProcess = new DynamicProcess(mi1, mi2, mi3);

    dynamicProcess.invoke();

    new AsynchMethodInvoker(dynamicProcess, "invoke", 1000);
  }
}
```

The `invoke` method of `DynamicProcess` returns a `Collection` containing the return values of all of the methods invoked. There is a placeholder within the collection for every method, even if the method does not return a value. This makes it easier to map the return value back to the method of the process. Here is an example:

```java
import java.util.Calendar;
import java.util.Collection;

import com.jmorgan.lang.AsynchMethodInvoker;
import com.jmorgan.lang.DynamicProcess;
import com.jmorgan.lang.MethodInvoker;

public class DynamicProcessExample {
  public static void main(String[] args) {
    Calendar c = Calendar.getInstance();
    c.set(1999, Calendar.DECEMBER, 31);

    MethodInvoker yearPointer = new MethodInvoker(c, "get", Calendar.YEAR);
    MethodInvoker monthPointer = new MethodInvoker(c, "get", Calendar.MONTH);
    MethodInvoker dayPointer = new MethodInvoker(c, "get", Calendar.DATE);

    DynamicProcess dynamicProcess = new DynamicProcess(yearPointer,
                                                       monthPointer,
                                                       dayPointer);

    Collection<?> returnValues = dynamicProcess.invoke();
    for (Object returnValue : returnValues)
      System.out.println(returnValue);
  }
}
```

You may still be wondering what advantages this brings. Before you toss the whole idea aside, consider the possibility that you have some kind of complex process that is unique per client. Let's also say the process is self-provisioned, and you create a way to store the process in a database using mnemonics mapped to the names of functions, and the client can decide the order of the methods for their implementation. In that scenario, `DynamicProcess` is not only useful, it is necessary!

Using Timer, TimerListener and TimerEvent

Offered as a replacement for `java.util.Timer` and `java.util.TimerTask` are the `Timer`, `TimerListener`, and the `TimerEvent` classes. I, personally, didn't like a few things about the out-of-the-box implementation of the `java.util.Timer` and `java.util.TimerTask` and chose another route. This, then, becomes a matter of preference, but this mechanism offers some advantages and less side-effects of the "bunching-up" that occurs with the built-in mechanism.

The most notable difference is the use of a `TimerListener`, which is an interface. This enables anything to be able to naturally participate within a timed implementation. The `Timer` class allows adding one or more `TimerListeners`, and offers many ways to optionally delay the first execution of the timer events and also ways to optionally invoke the events multiple times.

Each time the timer runs, it notifies all registered `TimerListener's` in their own thread, thus preventing the "bunching up" and rapid-firing of timed methods. This does mean, however, that when the `Timer` is set to run multiple times, the `TimerListener` implementation should ideally be able complete within the timed frequency, else the implementation of the event should be made thread safe. It takes common sense to realize that if you have a `TimerListener` that updates a cache of changes to a database every two seconds, it should not take two seconds to perform the update.

TimerListeners have a single method which receives a TimerEvent object containing information about the event:

```
public void timerEventTriggered(TimerEvent evt);
```

Here is a pretty complete example of the uses and features of this implementation within the DEF (this page and continuing onto the next):

```
import com.jmorgan.util.Timer;
import com.jmorgan.util.TimerEvent;
import com.jmorgan.util.TimerListener;

public class TimerExample {
   public static void main(String[] args) {
      Timer t = new Timer(2000, 5, "Hello");
      t.addTimerListener(new Listener("One"));
      t.addTimerListener(new Listener("Two"));
      t.addTimerListener(new Listener("Three"));
      t.addTimerListener(new Listener("Four"));
      t.addTimerListener(new Listener("Five"));
      t.start();

      new Timer(new Listener("1000"), 1000, 65);
      new Timer(new Listener("30000"), 30000, 3);
   }
}
```

```
class Listener implements TimerListener {
  String id;

  Listener(String id) { this.id = id; }

  public synchronized void timerEventTriggered(TimerEvent evt) {
    System.out.println(id + " Received event-> " + evt.getTime() +
                          ": " + evt.getMessage());
  }
}
```

Using AsynchMethodInvoker as a Timer

Another trick I frequently use when I need to ensure timed implementations can never step on top of one another is to use `AsynchMethodInvoker` to schedule a method, and then reschedule the method at the end of its invocation:

```
import com.jmorgan.lang.AsynchMethodInvoker;

public class AsynchMethodInvokerAsTimerExample {
  private static int maxRuns = 10;
  private static int currentRun = 0;

  public static void main(String[] args) {
    new AsynchMethodInvoker(AsynchMethodInvokerAsTimerExample.class,
                                                   "doUpdates", 1000);
  }

  public static void doUpdates() {
    System.out.println("Doing Updates");

    if (currentRun++ < maxRuns)
      new AsynchMethodInvoker(AsynchMethodInvokerAsTimerExample.class,
                                                   "doUpdates", 1000);
  }
}
```

Summary

Asynchronous processes can be canceled, and can also be easily restarted, which makes this framework quite unique and very convenient and powerful. There is more than one way to obtain the return value of an asynchronous call, by using a combination of `hasCompleted` and `getReturnValue`, or by using an `InvocationListener`.

Additionally, we see some use of the included `ThreadUtility` class. Though the DEF internally uses the `ThreadUtility`, if you have your own way of handling these things, please feel free to use what you like best.

Finally, we played with `DynamicProcess`, showing that any set of methods can be invoked dynamically or asynchronously. This proves to be very useful when you have the primitive methods you need for a given process, but cannot determine at development time the order you should run the primitives. The return values of all of the methods invoked by `DynamicProcess` are available from the `invoke` method. Furthermore, combined with `AsynchMethodInvoker`, you can run the entire dynamic process in a separate thread.

You can use a `Timer` to invoke or repeat timed events to `TimerListener`s. Or, you can use `AsynchMethodInvoker` to ensure you'll never have the timed event step on itself and eliminate all worries about thread-safe code.

Look at your code and discover if you have any need to cancel or restart asynchronous threads, or, if you have any implementations where using an `InvocationListener` would be better. Maybe you can identify opportunities to reduce code complexity with the use of `DynamicProcess`. While you do that, I'm going to put together Chapter 3.

Chapter 3 – Beans Support

The DEF includes special support for standard and non-standard Java beans. As you read about the features of this API, you may be wondering why you'd use this API rather than the popular BeanUtils API from Apache. I have no real problems with BeanUtils, except that there are some stronger features of this API, fewer classes to master, and a much simpler interface.

For the examples in this chapter, you will want to create a small text file named “test.txt”, and put it in the local directory where you are writing this code, or make the appropriate changes to the programs to ensure you are working with the file you create.

Creating Instances

The main class in this set is the `BeanService` utility class, and is the basis for the beans support within these utilities. All of the methods of this class are `static`, because of their general utility. To begin, there are four methods available to obtain an instance to a bean, either by using the fully qualified class name, or by using a reference to a class. They are:

```
public static Object getBean(String className);
public static Object getBean(Class<?> type);
```

These first two methods return an instance to a class. There is little surprising or special about these methods. They are simply part of the API to make it complete.

The next two methods, though, are special. These return an instance of a class using a constructor that takes the provided parameters. This way, you have a way of dynamically obtaining an instance even if the bean does not contain a no-argument constructor.

```
public static Object getBean(String className, Object...parameters);
public static Object getBean(Class<?> type, Object...parameters);
```

An example of the use of these methods is shown in the following section.

Obtaining Simple Property Values

For any dynamic bean utility, a robust ability to obtain property values is essential. There are many options available, making this next method quite capable of returning the value of just about any property. The method is:

```
public static final Object getPropertyValue(Object source, String expression);
```

This method returns the value of the given `source` bean's property defined by the given `expression`. To get a simple property, then, is pretty straightforward, as in this example (Note the use of the `getBean` method to obtain the instance of a `File` object):

```
import java.io.File;
import java.lang.reflect.InvocationTargetException;
import java.util.Date;

import com.jmorgan.beans.util.BeanService;

public class SimpleBeanPropertyExample {
  public static void main(String[] args) throws IllegalArgumentException,
                                                InstantiationException,
                                                IllegalAccessException,
                                                InvocationTargetException {
     File file = (File)BeanService.getBean(File.class, "test.txt");

     String fileName = (String)BeanService.getPropertyValue(file, "absolutePath");

     System.out.println("The absolute name of the file is: " + fileName);
  }
}
```

The first part of the example above uses the `getBean` method to show that a bean instance can be created using a constructor other than a no-argument constructor. Then, we see that the `getPropertyValue` method is smart enough to understand that the expression, “`absolutePath`”, resolves to a method `getAbsolutePath()`. It then invokes that method and returns the value.

Bean-compliant properties can be obtained in this way. That is, given the name of the property, the `getPropertyValue` method obtains the accessor method for *`propertyName`* by using either `getPropertyName` or `isPropertyName`, based upon the type of the property.

Obtaining Non Bean-Compliant Values

Many classes have properties or otherwise obtainable values that do not have bean compliant methods. For the `File` class, the `length`, `lastModified` and many other values do not have bean compliant method names, and for collections, there is the `size` method. For the DEF, it is no problem, as you can clearly see in the below example:

```
import java.io.File;
import java.util.Date;

import com.jmorgan.beans.util.BeanService;

public class SimpleBeanPropertyExample {
  public static void main(String[] args) {
    File file = new File("/");

    boolean exists = (Boolean)BeanService.getPropertyValue(file, "exists");
    boolean canRead = (Boolean)BeanService.getPropertyValue(file, "canRead");
    boolean canWrite = (Boolean)BeanService.getPropertyValue(file, "canWrite");
    boolean canExecute = (Boolean)BeanService.getPropertyValue(file,
                                                               "canExecute");
    long lastModifedDate = (Long)BeanService.getPropertyValue(file,
                                                               "lastModified");
    long length = (Long)BeanService.getPropertyValue(file, "length");

    System.out.println("The file exists: " + exists);
    System.out.println("We can read from the file: " + canRead);
    System.out.println("We can write to the file: " + canWrite);
    System.out.println("We can execute the file: " + canExecute);
    System.out.println("The last modified date of the file is: " +
                                          new Date(lastModifedDate));
    System.out.println("The length of the file is: " + length);
  }
}
```

The `BeanService.getPropertyValue` method is smart enough to find the accessor method, if at all possible, and return its value. If no method closely related to the property exists, a `RuntimeException` wrapped around a `NoSuchMethodException` is thrown.

Obtaining Indexed Values – Values of Arrays or Lists

Sometimes the return value of a method is an array or a list. In this case, you can obtain the entire array or list, or you can elect to grab one element, a slice of them, or selected elements. Note, you may have to modify the next example a little for it to work, but I've attempted to keep things so they will work for most:

```
import com.jmorgan.beans.util.BeanService;

public class BeanArrayPropertyValue {
  public static void main(String[] args)  {
    File file = new File("/");

    File[] files = (File[]) BeanService.getPropertyValue(file, "listFiles");
    for (File f : files)
      System.out.println("List: " + f.getAbsolutePath());

    File fourthFile = (File)BeanService.getPropertyValue(file, "listFiles[3]");
    System.out.println("Fourth File By Index: " + fourthFile.getAbsolutePath());

    Object[] slice =
      (Object[])BeanService.getPropertyValue(file, "listFiles[2-5]");

    for (Object f : slice)
      System.out.println("Slice: " + ((File)f).getAbsolutePath());

    Object[] pickAndChoose = (
      Object[])BeanService.getPropertyValue(file, "listFiles[1, 3, 4, 6]");

    for (Object f : pickAndChoose)
      System.out.println("Pick And Choose: " + ((File)f).getAbsolutePath());

    Object[] combo = (Object[])BeanService.getPropertyValue(file,
                           "listFiles[1, 3-5, 7]");
    for (Object f : combo)
      System.out.println("Combo: " + ((File)f).getAbsolutePath());

  }
}
```

So, this is the way it works. When a single number is enclosed in brackets, such as in the above `[3]`, the method returns the single value at that index. If two numbers are enclosed in brackets separated by a dash, `[2-5]`, it returns an array of values containing the elements from the first index through to the second index, inclusively. Two or more comma delimited numbers enclosed in brackets, such as in `[1, 3, 4, 6]`, is interpreted as a list of individual index values, and an array of values is returned containing the values at each of the index locations. You can also combine the syntax, with a mixture of comma delimited numbers and/or ranges, `[1, 3-5, 7]`, and get an array containing those elements.

A couple of other things to note. When selecting a single index value, a single object reference is returned. When selecting a range or more than one index value, an `Object` array is returned. If the raw property value resolves to a `String`, `StringBuffer`, or `StringBuilder`, then you are indexing to individual characters or character ranges within that `String`, `StringBuffer`, or `StringBuilder`, and will receive back either a single character if only a single index is used, and an array of Objects containing the characters of the slices and/or indexes.

Ranges must be in `min - max` format. Individual indexes, however, do not have to be in sequential order, and therefore can be returned in an arbitrary order, such as `[8, 3-5, 9, 11-14, 0]` to receive an object array with the values from the original array from indexes, `8, 3, 4, 5, 9, 11, 12, 13, 14`, and `0`, in that order.

As you might can imagine, we just opened up a 55 gallon drum of things that can go wrong here. Essentially, the rules are much like if you were not using this API and working directly against an array of values referencing indices there. Certainly, `ArrayIndexOutOfBoundsException`s can occur.

Accessing Nested Values

Nested properties can be obtained by separating the properties with periods as in the standard Java beans compliant way, and, of course, not all properties need to be fully bean compliant:

```
import java.io.File;

import com.jmorgan.beans.util.BeanService;

public class NestedPropertyExample {
  public static void main(String[] args) {
    File file = new File("test.txt");

    String parentFileName = (String)BeanService.getPropertyValue(file,
                                     "absoluteFile.parent");

    System.out.println("Parent File: " + parentFileName);

    int parentFileLength = (Integer)BeanService.getPropertyValue(file,
                                     "absoluteFile.parent.length");

    System.out.println("Parent File Length: " + parentFileLength);

    int length = (Integer)BeanService.getPropertyValue(new File("\\"),
                                     "listFiles[5].toString.length");
    System.out.println("Fifth File Name Length: " + length);
  }
}
```

The same kinds of problems you'd experience coding this normally can occur here too. The most common exceptions you'll encounter with nesting are `NullPointerException` and `ClassCastException`., but you may also encounter an `ArrayIndexOutOfBoundsException` if using index values or ranges out of bounds.

Using Expressions

It is possible to construct simple expressions on properties as well. Simple mathematical operations and string concatenation is possible. I decided not to try to develop a full expression parser and essentially a full-blown sub-language, because the base Java language is better equipped for that, and we're already taxing performance, but some basic expressions are possible:

```java
import java.io.File;

import com.jmorgan.beans.util.BeanService;

public class SimpleExpressionsExample {
   public static void main(String[] args) {
      File file = new File("test.txt");

      String concatenated = (String)BeanService.getPropertyValue(file,
                                    "absoluteFile.parent + '\\' + name");

      System.out.println("Concatenated String: " + concatenated);

      double length = (Double)BeanService.getPropertyValue(file,
                              "name.length + absoluteFile.parent.length");

      System.out.println("Combined Length: " + length);
   }
}
```

The mathematical operators are:

+	To add two numbers or concatenate Strings
–	To subtract two numbers (unless it is used as a range separator in an array slice expression)
*	To multiply two numbers
/	To divide two numbers
%	Modulo division
^	Raise the first number to the power of the second

All mathematical operations are performed with `double` precision, and are returned as a `Double`.

Other BeanService Methods

There are some safety measures that might help you determine if something you are about to do will be successful. For some objects, these "safety" measures can cause more damage than good. Other methods can help with generating a `hashCode` for a class, or writing an object's properties to a `String`. Other available features are:

```java
public static Class<?> getPropertyType(Object source, String expression);
```

This method will return the type of a property or expression. Any expression accepted by *`getPropertyValue`* is allowed here. The only real notes of concern with this is that when using array expressions that return more than one value, the type will always be an `Object` array. When any mathematical expression other than String concatenation is evaluated, it will be `Double`.

```java
public static final Method[] getBeanAccessors(Object bean);
```

Returns a best-guess list of methods that appear to return information about a class. Basically, this method returns all methods having no arguments and non-`void` return types. Be careful, though, because not all methods of a class fitting this description are actually accessors. For example, running this method against a `File` object returns the methods `delete`, `createNewFile`, `mkdir`, and `mkdirs`. Certainly, there is no way to establish a perfect list of accessors, since folks that write Java classes don't always adhere to JavaBean compliant practices, not even the great people of Sun.

```
public static final Method[] getBeanMutators(Object bean);
```

Returns a best-guess list of methods beginning with the "`set`" prefix and taking at least one argument. This cannot guarantee that every method actually mutates the state of an object, but the general contract of a "`set`" method is assumed to do so.

```
public static ArrayList<String> getPropertyNames(Object bean);
```

Returns a best-guess list of properties based upon the list of methods returned by `getBeanAccessors`. If an accessor method name begins with "`get`" or "`is`", this is stripped from the name to assume the name of the property. Not all properties are mutable, so the goal of this method is to return the names of properties that provide information about the bean. The same warning applies here as the one for `getBeanAccessors`. For `File` objects, this method returns `delete`, `createNewFile`, `mkdir`, and `mkdirs`. The fully dynamic nature of this feature can set you up for problems if you don't know what it is you are running against.

```
public static ArrayList<String> getMutablePropertyNames(Object bean);
```

Returns the best-guess list of bean-compliant mutable properties of a bean. This method assumes that methods beginning with the prefix "`set`" and having one or more parameters is a mutator for a property, though there is no guarantee methods following this standard actually mutate the state of the `bean`, nor will it return the names of properties that are modifiable via a method not having bean-compliant names.

```
public static final String toString(Object bean, String delimiter);
```

This method renders the given `bean` into a printable String by obtaining the `bean`'s accessors and then concatenating their values using the given `delimiter`. This is by no means a replacement for a developer to not develop a proper `toString` method for their beans, but is very useful when you need to render a bean or object into a string when that bean or object does not contain a proper method.

```
public static final int getHashCode(Object bean);
```

Creates a hash code using the current state of the `bean`. This is useful for classes with no or poorly written `hashCode` methods, or even as a convenient replacement for you having to develop your own `hashCode` method. However, there is little chance that if the class has a properly written `hashCode` method, that this method will resolve to the same hash code. However, this method will return a hash code unique to the state of the `bean` such that it is usable with hashing mechanisms.

Setting Property Values

Property values can be set in a very similar way as getting values. Values can be set using all expressions supported for getting values provided the expressions actually resolve to mutable values. Obviously, then, mathematical and concatenation expressions will not resolve to mutable properties.

Likewise, care should be taken when mutating values of array slices and multiple array values, as unexpected results can easily occur.

Note this method is designed to receive a single value for the property to set. Therefore, for methods requiring multiple values, defer to using `MethodInvoker`.

It is best to keep things simple, or by obtaining the values and mutating them individually. Here is a simple example:

```java
import java.util.Calendar;

import com.jmorgan.beans.util.BeanService;

public class SetBeanPropertyExample {
   public static void main(String[] args) {
      Calendar c = Calendar.getInstance();

      System.out.println("Calendar Value: " + c.getTime());

      BeanService.setPropertyValue(c, "firstDayOfWeek", 3);
      BeanService.setPropertyValue(c, "lenient", false);
      BeanService.setPropertyValue(c, "clear", Calendar.MONTH);

      System.out.println("First Day Of Week: " + c.getFirstDayOfWeek());
      System.out.println("Is Lenient: " + c.isLenient());
      System.out.println("Calendar Value: " + c.getTime());
   }
}
```

As you can see, both Java Bean compliant properties can be set, as well as non-Java Bean compliant property names, provided the methods matching those names receive a single parameter and, of course, actually exist.

Using BeanComparator

If you are a Commons BeanUtils user, you're really going to think I just copied what they did. I promise, this `BeanComparator` is similar to the Commons BeanUtils BeanComparator class in name only, and you'll soon see this one is much more sophisticated. Using this BeanComparator, you're likely to never need to write another comparator, ever!

As the name of the class suggests, this class can compare any two beans based upon one or more properties. The "properties" to compare are expressions with the same rules as that for the `BeanService.getPropertyValue` method detailed earlier in this chapter. Therefore, bean compliant, non-bean compliant, nested properties, array elements, and even results of expressions can be compared between two beans. Descending sorts can be obtained by prefixing the property name or expression with a minus sign, as in `"-name"`. Furthermore, unlike Commons BeanUtils BeanComparator, this class supports the definition of more than one property.

Here is an example:

```
import java.io.File;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

import com.jmorgan.beans.util.BeanService;
import com.jmorgan.util.comparator.BeanComparator;

public class BeanComparatorExample {
   public static void main(String[] args) {
      File file = new File("/");
      File[] files = file.listFiles();

      List<File> fileList = Arrays.asList(files);

      BeanComparator<File> comparator = new BeanComparator<File>();
      comparator.addCompareProperty("name");

      sortAndShow(fileList, comparator, "**** Sorted by name ****", "name");

      comparator.clearCompareProperties();
      comparator.addCompareProperty("length");

      sortAndShow(fileList, comparator, "\n**** Sorted by length ****", "length");

      comparator.addCompareProperty("name");

      sortAndShow(fileList, comparator, "\n**** Sorted by length then name ****", "length");

      comparator.clearCompareProperties();
      comparator.addCompareProperties("-length", "name");

      sortAndShow(fileList, comparator,
         "\n**** Sorted by length descending, then by name ****", "length");
   }

   private static void sortAndShow(List<File> files,
                                   BeanComparator<File> comparator,
                                   String title, String property) {
      Collections.sort(files, comparator);
      System.out.println(title);
      for (File file : files) {
         System.out.println("\t" + file.getName() + " - " +
                            BeanService.getPropertyValue(file, property));
      }
   }
}
```

The first note about the example is that `BeanComparator` uses the parameterized type of the bean to be compared. After construction, you add one or more compare properties through the `addCompareProperty` and `addCompareProperties` methods. `BeanComparator` also has a couple of convenience constructors enabling you to set compare properties during instantiation:

```
public BeanComparator(Collection<String> properties);
public BeanComparator(String[] properties);
```

The `clearCompareProperties` method allows you to reset an existing `BeanComparator` instance for another use. Two other methods enable you to either set or reset the properties of the `BeanComparator` instance:

```
public void setCompareProperties(Collection<String> compareProperties);
public void setCompareProperties(String[] compareProperties);
```

Using AnyObjectComparator

This class can be used to reasonably compare any two objects, even if they do not implement the `Comparable` interface:

```
import com.jmorgan.util.comparator.AnyObjectComparator;

public class AnyObjectComparatorExample {
  public static void main(String[] args) {
    AnyObjectComparator c = new AnyObjectComparator();

    Boolean b1 = new Boolean(true);
    Boolean b2 = new Boolean(true);
    System.out.println("Boolean Compare: " +
                       c.compare(b1, b2));

    Character c1 = new Character('A');
    Character c2 = new Character('B');
    System.out.println("Character Compare: " +
                       c.compare(c1, c2));

    Integer integerNumber = new Integer(4);
    Long longNumber = new Long(4);

    System.out.println("Different Number types Compare: " +
                       c.compare(longNumber, integerNumber));

    StringBuilder sb = new StringBuilder("King Kong");
    System.out.println("String/StringBuilder types Compare: " +
                       c.compare(sb, "King Kong"));

    c.setNullHandling(AnyObjectComparator.NULLS_ARE_FIRST);
    System.out.println("String to null comparison where nulls are first: " +
                       c.compare("Godzilla", null));

    c.setNullHandling(AnyObjectComparator.NULLS_ARE_LAST);
    System.out.println("String to null comparison where nulls are last: " +
                       c.compare("Godzilla", null));

    c.setNullHandling(AnyObjectComparator.NULLS_ARE_NULL);
    System.out.println("String to null comparison where nulls are null: ");
    System.out.println(c.compare("Godzilla", null));
  }
}
```

The algorithm, though relatively simple, makes every possible attempt to create a successful comparison. First, if both objects are `null`, they are considered equal. If either object is `null`, then the evaluation is determined by the `nullHandling` property of the class, where `null` values can be considered as first or last. The default is that `null`s are first.

If both objects implement the `Comparable` interface, then it attempts `object1.compareTo(object2)`. This could cause a `ClassCastException`, which is not fatal to the process. If both objects are some kind of `Number`, it compares the `doubleValue` of each of the two numbers and returns the result of the comparison. Not all `Number` subclasses implement `Comparable`, and even when they do, `Long` to `Integer` comparisons will fail, even if they can be reasonably compared, so this is a useful feature.

It then attempts a combination of `object1.equals(object2)` and `object2.equals(object1)`, and if either returns true, it considers them equal. It does both because there is no way to know how a given instance compares itself to another, and one comparison may return `false`, whereas another may return `true`. As a last resort, it returns `object1.toString().compareTo(object2.toString())`.

There is one other constructor for `AnyObjectComparator` allowing you to define `null` handling, though you can always set `null` handling later with `setNullHandling`:

```
public AnyObjectComparator(int nullHandling);
```

Summary

In this chapter, you've seen quite a number of elements of the DEF that support standard, and even non-standard Java Beans. We've seen that we can create instances of classes, even if they do not have a no-argument constructor. Additionally, we learned that we can access properties, even if the bean is not necessarily bean-compliant. Complex properties, including nested properties and properties of a bean that are elements of an `ArrayList` or `Array` are available as well.

The DEF even allows basic expressions to be created against objects, giving elemental mathematical expressions or String concatenation features. Using the same methods available for obtaining values (excepting expressions, of course) you can set property values as well.

We learned there are many ways to obtain information about beans by obtaining its accessors and mutators, and an ability to create complex Comparators and use the `AnyObjectComparator` to safely compare or sort any two objects.

Chapter 4 – Collections Extensions

The DEF contains a number of extensions to the Collections API. These extensions support a number of capabilities that can simplify selections and perform other common tasks.

As a segue from the previous chapter, we'll start with the `CollectionComparator` class:

Using CollectionComparator

`CollectionComparator` is a class providing a reusable way to compare collections:

```
import java.util.ArrayList;

import com.jmorgan.util.comparator.CollectionComparator;

public class CollectionComparatorExample {
   public static void main(String[] args) {
      ArrayList<String> collection1 = new ArrayList<String>();
      ArrayList<String> collection2 = new ArrayList<String>();

      String[] filler = { "One", "Two", "Three", "Four" };
      for (String s : filler) {
         collection1.add(s);
         collection2.add(s);
      }

      CollectionComparator<String> c = new CollectionComparator<String>();

      System.out.println("Test1: Return Value of Compare: " +
                         c.compare(collection1, collection2));

      collection1.add("Five");

      System.out.println("Test2: Return Value of Compare: " +
                         c.compare(collection1, collection2));

      collection2.add("Five");

      System.out.println("Test3: Return Value of Compare: " +
                         c.compare(collection1, collection2));

      collection2.add("Six");

      System.out.println("Test4: Return Value of Compare: " +
                         c.compare(collection1, collection2));
   }
}
```

This `Comparator`'s `compare` algorithm first starts with `null` handling. If both collections are `null`, they are considered equal, otherwise `null` is considered less than a non- `null` collection. If both collections are empty, they are considered equal. If both collections have the same number of elements and contain the same elements, they are, of course, considered equal. If one collection has fewer elements than the other and the other contains all the elements of the one, then the one with fewer elements is considered less than the other.

Following that, the unique elements of both collections are extracted, sorted and then compared side by side, using the `AnyObjectComparator` detailed in the previous chapter. The process returns the comparison result of the first non-equal elements.

The number of elements compared is never more than the number of elements in the smaller sized collection. Therefore, if all of the unique elements of the smaller sized collection are equal to those on the larger sized collection, the method returns -1 if the first collection is the smaller sized collection, otherwise it returns 1, indicating the first collection is the larger sized of the two collections.

Using Collection Services

There are three primary collection services within the DEF:

- `CollectionSelector` – This service selects elements from a collection based upon a defined algorithm. We will show examples of some of the existing implementations, and provide brief descriptions of the remaining ones.
- `CollectionExecutor` – Executes methods on every element within a collection. Existing implementations include the `ThreadExecutor`, and the `MethodExecutor`.
- `CollectionAggregator` – Provides a means to aggregate on elements of a collection via one or more aggregation functions. Existing aggregation functions include `NumericalAggregator` and `CountAggregator`.

CollectionSelector

The `CollectionSelector` is an abstract class fundamentally designed to loop through a collection and choose elements by implementations of the `isElementSelected` method. `CollectionSelector` is a typed class, designating the type of elements contained by the collection. `CollectionSelector`s also enable the programmer to control the maximum number of elements returned.

When using a `CollectionSelector`, once it is provided a reference to a collection, an invocation of `getSelectedElements` returns those elements of the collection, if any, matching the semantic of the implementation.

Creating an implementation of a `CollectionSelector` is very easy. Here is an implementation that returns all non-directory files from a collection:

```
import java.io.File;
import java.util.Arrays;
import java.util.Collection;
import java.util.List;

import com.jmorgan.util.collection.CollectionSelector;

public class FileCollectionSelector extends CollectionSelector<File> {
  public FileCollectionSelector(Collection<? extends File> collection)
                                        throws NullPointerException {
    super(collection);
  }

  // The one and only method you need to implement
  protected boolean isElementSelected(File element) {
    return !element.isDirectory();
  }

  public static void main(String[] args) {
    File file = new File("/");

    File[] files = file.listFiles();
    List<File> fileList = Arrays.asList(files);

    FileCollectionSelector selector = new FileCollectionSelector(fileList);

    // If your root directory only contains directories,
    // this will return an empty collection
    Collection<File> trueFiles = selector.getSelectedElements();

    for (File trueFile : trueFiles) {
      System.out.println(trueFile.getAbsolutePath());
    }
  }
}
```

As you can see, the only method you need to implement is `isElementSelected` which should return `true` if the element matches the semantic of the selector, or `false` if not.

Using InstanceSelector

`InstanceSelector` is an existing selector implementation useful on untyped collections or typed collections that may contain subclasses of many types. Here is an example:

```
import com.jmorgan.util.collection.InstanceSelector;

public class InstanceSelectorExample {
      public static void main(String[] args) {
    java.util.ArrayList list = new java.util.ArrayList();

    list.add("String1");
    list.add(new Thread("Thread1"));
    list.add(new Thread("Thread2"));
    list.add("String2");
    list.add(new StringBuilder("StringBuilder1"));
    list.add("String3");
    list.add(new StringBuilder("StringBuilder2"));
    list.add("String4");
    list.add(new StringBuilder("StringBuilder3"));
    list.add(new Thread("Thread3"));
    list.add(new StringBuilder("StringBuilder4"));
    list.add("String5");
    list.add(new Thread("Thread4"));
    list.add(new Thread("Thread5"));
    list.add("String6");
    list.add(new StringBuilder("StringBuilder5"));
    list.add(new StringBuilder("StringBuilder6"));
    list.add(new Thread("Thread6"));

    InstanceSelector selector = new InstanceSelector(list, String.class);
    System.out.println(selector.getSelectedElements());

    selector.setInstanceType(StringBuilder.class);
    System.out.println(selector.getSelectedElements());

    selector.setInstanceType(Thread.class);
    System.out.println(selector.getSelectedElements());
  }
}
```

Note the ability to change the instance type and reselect.

EqualsSelector

The `EqualsSelector` selects elements of a collection equal to a “tester” value:

```
import java.util.ArrayList;

import com.jmorgan.util.collection.EqualsSelector;

public class EqualsSelectorExample {
  public static void main(String[] args) {
     ArrayList<String> list = new ArrayList<String>();
     list.add("One");
     list.add("One");
     list.add("Two");
     list.add("One");
     list.add("Two");
     list.add("Three");
     list.add("One");
     list.add("Two");
     list.add("Three");
     list.add("Four");
     list.add("One");
     list.add("Two");
     list.add("Three");
     list.add("Four");
     list.add("Five");

     EqualsSelector<String> selector = new EqualsSelector<String>(list, "One");
     System.out.println(selector.getSelectedElements());

     selector.setTester("Two");
     System.out.println(selector.getSelectedElements());

     selector.setTester("Three");
     System.out.println(selector.getSelectedElements());

     selector.setTester("Four");
     System.out.println(selector.getSelectedElements());

     selector.setTester("Five");
     System.out.println(selector.getSelectedElements());

     selector.setTester("Six");
     System.out.println(selector.getSelectedElements());
  }
}
```

The mechanism for comparing values is via the `equals` method of each element against the tester. That is, its `isElementSelected` method is:

```
  protected boolean isElementSelected(E element) {
    return element.equals(this.tester);
  }
```

DifferenceSelector

The `DifferenceSelector` selects elements of a collection based upon their comparison to a “tester” element and an optionally supplied `Comparator`. If a `Comparator` isn't supplied, the “tester” will be used for comparing against the elements of the collection. There are several modes for selection, defined by constants of the class:

```
import java.util.ArrayList;
import java.util.Comparator;

import com.jmorgan.util.collection.DifferenceSelector;

public class DifferenceSelectorExample {
  private static class TestComparator implements Comparator {
    public int compare(Object o1, Object o2) {
      System.out.println("Testing " + o1 + " vs " + o2);
      return (((String)o1).compareTo((String)o2));
    }
  }

  public static void main(String[] args) {
    ArrayList<String> list = new ArrayList<String>();
    list.add("One");
    list.add("One");
    list.add("Two");
    list.add("One");
    list.add("Two");
    list.add("Three");
    list.add("One");
    list.add("Two");
    list.add("Three");
    list.add("Four");
    list.add("One");
    list.add("Two");
    list.add("Three");
    list.add("Four");
    list.add("Five");

    System.out.println("Comparable");
    DifferenceSelector<String> selector =
                       new DifferenceSelector<String>(list, "One",
                                  DifferenceSelector.SELECT_IF_LESS_THAN);
    System.out.println(selector.getSelectedElements());

    selector.setCompareType(DifferenceSelector.SELECT_IF_GREATER_THAN);
    System.out.println(selector.getSelectedElements());

    selector.setCompareType(DifferenceSelector.SELECT_IF_EQUAL_TO);
    System.out.println(selector.getSelectedElements());

    System.out.println("Comparator");
    TestComparator cmp = new TestComparator();
    selector.setComparison(cmp, "One");
    selector.setCompareType(DifferenceSelector.SELECT_IF_LESS_THAN);
    System.out.println(selector.getSelectedElements());

    selector.setCompareType(DifferenceSelector.SELECT_IF_GREATER_THAN);
    System.out.println(selector.getSelectedElements());

    selector.setCompareType(DifferenceSelector.SELECT_IF_EQUAL_TO);
    System.out.println(selector.getSelectedElements());
  }
}
```

If using the tester, then the tester must implement `Comparable`. The comparison is conducted with the tester's `compareTo` method against each element of the collection. Therefore, if the selection mode is set to `DifferenceSelector.SELECT_IF_EQUAL_TO`, then the selector returns all elements equal to the tester. If the mode is set to `DifferenceSelector.SELECT_IF_LESS_THAN`, then the selector returns all elements less than the tester, and if the mode is `DifferenceSelector.SELECT_IF_GREATER_THAN`, then the selector returns all elements greater than the tester.

PropertyValueSelector

By far, the most powerful and flexible of the existing implementations is the `PropertyValueSelector`. This selector digs deep into the `BeanService` to compare values of properties of the elements within the collection by operating against one or more property value maps. Each property value map can be independently compared based upon one of many comparison modes. The comparison modes are defined as constants in the `PropertyValueSelector` class.

Property names are actually expressions that allow nesting, and, in fact, any expression allowed by the `BeanService.getPropertyValue` method. Here is a simple example:

```
import java.io.File;
import java.util.Arrays;
import java.util.List;

import com.jmorgan.util.collection.PropertyValueSelector;

public class PropertyValueSelectorExample {
  public static void main(String[] args) {
    File file = new File("..");
    File[] files = file.listFiles();
    List<File> fileList = Arrays.asList(files);

    System.out.println(fileList);

    PropertyValueSelector<File> selector =
                         new PropertyValueSelector<File>(fileList);
    selector.addPropertyValueMap("name", "M",
                         PropertyValueSelector.IS_GREATER_THAN);
    System.out.println(selector.getSelectedElements());

    selector.addPropertyValueMap("directory", false);
    System.out.println(selector.getSelectedElements());

    selector.addPropertyValueMap("length", 1000,
                         PropertyValueSelector.IS_LESS_THAN);
    System.out.println(selector.getSelectedElements());
  }
}
```

Depending upon your file system, the above example may need some changing, but, in general, the example demonstrates the fundamental use of the selector.

The `PropertyValueSelector` supports the following comparisons:

IS_EQUAL_TO
IS_NOT_EQUAL_TO
IS_GREATER_THAN
IS_LESS_THAN
IS_GREATER_THAN_OR_EQUAL_TO
IS_LESS_THAN_OR_EQUAL_TO
IS_IN
IS_NOT_IN
MATCHES
NOT_MATCHES
IS_NULL
IS_NOT_NULL

All comparisons work somewhat intuitively. The IS_IN and IS_NOT_IN comparison types, however, work against collections or arrays of values. That is to say if the value of the property or exception "is in" or "is not in" a given collection or array.

IS_NULL and IS_NOT_NULL are just concerned if the property or expression resolves to NULL.

MATCHES and NOT_MATCHES compares the value of the property or exception to a provided regular expression. In these comparison types, NULL is considered to match to NULL. Otherwise the value of the property or exception is first converted to a String via the object's `toString` method, and then matched to the regular expression.

IndexSelector

There are times when you need to operate on subsets of a collection based upon an index range. The `IndexSelector` provides the convenience of returning those elements in that range. The range can be set during construction, or later, with `setBeginIndex` and `setEndIndex`. Here is an example:

```
import java.util.ArrayList;
import java.util.Collections;

import com.jmorgan.util.collection.IndexSelector;

public class IndexSelectorExample {
  public static void main(String[] args) {
    ArrayList<String> collection = new ArrayList<String>(100);

    for (int i = 0; i < 100; i++)
      collection.add("" + i);

    System.out.println("Selecting elements 10 to 20");
    IndexSelector<String> indexSelector = new IndexSelector<>(collection, 10, 20);
    for (String s : indexSelector.getSelectedElements())
      System.out.println(s);
    System.out.println("\nSelecting elements 32 to 45");
    indexSelector.setBeginIndex(32);
    indexSelector.setEndIndex(45);
    for (String s : indexSelector.getSelectedElements())
      System.out.println(s);

    System.out.println("\nSelecting elements 76 to 62");
    indexSelector.setBeginIndex(76);
    indexSelector.setEndIndex(62);
    for (String s : indexSelector.getSelectedElements())
      System.out.println(s);
  }
}
```

RegexSelector

One more existing implementation exists, the `RegexSelector`, which returns values from a collection of strings matching a provided regular expression. It is relatively simple and straightforward:

```
import java.util.ArrayList;
import com.jmorgan.util.collection.RegexSelector;

public class RegexSelectorExample {
  public static void main(String[]_args) {
    ArrayList<String> list = new ArrayList<String>();
    list.add("this is a test");
    list.add("King Kong loves Godzilla");
    list.add("Nobody saw me do it, no one can prove a thing.");
    list.add("The brown fox jumped over the moon");
    list.add("Today, December 7th, is a day that will live in infamy");
    list.add("Long ago, in a galaxy far far away");
    list.add("Be quiet.  Do you smell that?");
    list.add("We're going to need a bigger boat");

    System.out.println("Should be all: " +
          new RegexSelector(list).getSelectedElements());
    System.out.println("All containing the letter 'v': " +
          new RegexSelector(list, "[Vv]").getSelectedElements());
    System.out.println("All containing the word 'is': " +
          new RegexSelector(list, "\\bis\\b").getSelectedElements());
    System.out.println("All containing the word 'is' or 'it': " +
          new RegexSelector(list, "\\bi[st]\\b").getSelectedElements());
    System.out.println("All containing words beginning with 'f': " +
          new RegexSelector(list, "\\bf\\w+").getSelectedElements());
    System.out.println("All containing 'on' in a word: " +
          new RegexSelector(list, "\\Bon\\B").getSelectedElements());
    System.out.println("No matches: " +
          new RegexSelector(list, "zzzzz").getSelectedElements());
  }
}
```

CollectionExecutor

The `CollectionExecutor` is fundamentally coded to loop through all of the elements of a collection and execute something, ideally a method of the element, but it could do anything. `CollectionExecutor` is a typed class, designating the type of elements contained by the collection. Like `CollectionSelector`, this class is abstract. Also like `CollectionSelector`, subclasses need only implement the **`protected abstract void`** `processElement(E element);` method.

Once a `CollectionExecutor` is provided a reference to a collection, an invocation of `execute`, starts the iteration process, and the `processElement` method performs some function related to the semantic of the implementation.

Creating an implementation of a `CollectionExecutor` is very easy. Here is an implementation that clears all the time values of a collection of `Calendar` objects:

```
import java.util.ArrayList;
import java.util.Calendar;
import java.util.Collection;

import com.jmorgan.util.ThreadUtility;
import com.jmorgan.util.collection.CollectionExecutor;

public class ClearTimeValuesExecutor extends CollectionExecutor<Calendar> {
  public ClearTimeValuesExecutor(Collection<? extends Calendar> collection)
                                        throws NullPointerException {
    super(collection);
  }

  protected void processElement(Calendar element) {
    element.clear(Calendar.MILLISECOND);
    element.clear(Calendar.SECOND);
    element.clear(Calendar.MINUTE);
  }

  public static void main(String[] args) {
    ArrayList<Calendar> listOfDates = new ArrayList<Calendar>();

    for (int i = 0; i < 10; i++) {
      Calendar c = Calendar.getInstance();
      System.out.println("Created: " + c.getTime());
      listOfDates.add(c);
      ThreadUtility.sleep(1000);
    }

    ClearTimeValuesExecutor executor = new ClearTimeValuesExecutor(listOfDates);
    executor.execute();

    for (Calendar c : listOfDates) {
      System.out.println(c.getTime());
    }
  }
}
```

As you can see, if it wasn't for the `main` method created in the example to demonstrate the functionality of the implementation, the entirety of the class is just a few lines of code. There are two pre-existing executor implementations of the `CollectionExecutor` class. The first is quite obviously desired.

ThreadExecutor

The `ThreadExecutor` does much as it sounds, starts a thread. It actually is a little more flexible than that. `ThreadExecutor` takes a collection of things that implement `Runnable`, and since `Thread` implements `Runnable`, it makes things quite convenient.

The process loops through the collection. If the instance is a `Thread`, it starts the `Thread`. If the instance is not a `Thread`, but only an implementation of `Runnable`, it creates a `Thread` associating it with the `Runnable` instance, and starts the `Thread`:

```java
import java.util.ArrayList;

import com.jmorgan.util.collection.ThreadExecutor;

public class ThreadExecutorExample implements Runnable {
   private String name;

   public ThreadExecutorExample(String name) {
      this.name = name;
   }

   public void run() {
      System.out.println(name);
   }

   public static void main(String[] args) {
     ArrayList<Runnable> list = new ArrayList<Runnable>();

     list.add(new ThreadExecutorExample("One"));
     list.add(new Thread(new ThreadExecutorExample("TOne")));
     list.add(new Thread(new ThreadExecutorExample("TTwo")));
     list.add(new ThreadExecutorExample("Two"));
     list.add(new Thread(new ThreadExecutorExample("TThree")));
     list.add(new ThreadExecutorExample("Three"));
     list.add(new Thread(new ThreadExecutorExample("TFour")));
     list.add(new Thread(new ThreadExecutorExample("TFive")));
     list.add(new ThreadExecutorExample("Four"));
     list.add(new ThreadExecutorExample("Five"));
     list.add(new Thread(new ThreadExecutorExample("TSix")));

     ThreadExecutor threadExecutor = new ThreadExecutor(list);
     threadExecutor.execute();
   }
}
```

So, either a `Thread` or a `Runnable`, when the `ThreadExecutor` runs, each is handled as expected.

MethodExecutor

The `MethodExecutor` seems redundant by name, since the intent of a `CollectionExecutor` is to provide a means to execute something in or with each element of the collection. This class, though, utilizes the `AsynchMethodInvoker` to asynchronously run a named method with an optional list of parameters on all elements of the collection:

```
import java.util.ArrayList;
import java.util.Calendar;

import com.jmorgan.util.ThreadUtility;
import com.jmorgan.util.collection.MethodExecutor;

public class MethodExecutorExample {
   public static void main(String[] args) {
      ArrayList<Calendar> listOfDates = new ArrayList<Calendar>();

      for (int i = 0; i < 10; i++) {
         Calendar c = Calendar.getInstance();
         System.out.println("Created: " + c.getTime());
         listOfDates.add(c);
         ThreadUtility.sleep(1000);
      }

      MethodExecutor<Calendar> methodExecutor =
                        new MethodExecutor<Calendar>(listOfDates, "set",
                                                     Calendar.SECOND, 0);
      methodExecutor.execute();

      for (Calendar c : listOfDates)
         System.out.println(c.getTime());
   }
}
```

Keep in mind, too, that since the `MethodExecutor` uses `AsynchMethodInvoker` to invoke the method, function pointers in the form of `MethodInvoker` can be used to establish the values of the parameters at the time the method is actually invoked. This makes for a very powerful and flexible collection of tools.

A Use Case for Combining Selectors and Executors

It becomes extremely convenient to utilize selectors when you need to isolate elements for an executor. This is especially necessary to ensure your selection will be able to perform the needed function. In the next example, a collection contains a list of `Runnable`s. You want to start any threads in the list. Here's how:

```java
import java.util.ArrayList;
import java.util.Collection;

import com.jmorgan.util.collection.InstanceSelector;
import com.jmorgan.util.collection.MethodExecutor;

public class SelectorExecutorUseCaseExample {
  public static void main(String[] args) {
    ArrayList<Runnable> list = new ArrayList<Runnable>();

    list.add(new SomeThread("Thread1"));
    list.add(new SomeRunnable("Runnable1"));
    list.add(new SomeRunnable("Runnable2"));
    list.add(new SomeThread("Thread2"));
    list.add(new SomeThread("Thread3"));
    list.add(new SomeRunnable("Runnable3"));
    list.add(new SomeThread("Thread4"));
    list.add(new SomeThread("Thread5"));
    list.add(new SomeRunnable("Runnable4"));
    list.add(new SomeRunnable("Runnable5"));
    list.add(new SomeThread("Thread6"));
    list.add(new SomeRunnable("Runnable6"));

    InstanceSelector selector = new InstanceSelector(list, Thread.class);
    Collection<?> threads = selector.getSelectedElements();

    MethodExecutor<Thread> executor =
      new MethodExecutor<Thread>((Collection<? extends Thread>) threads,
                                                                    "start");
    executor.execute();
  }
}

class SomeThread extends Thread {
  SomeThread(String threadName) {
    super(threadName);
  }

  public void run() {
    System.out.println("I am thread " + this.getName());
  }
}

class SomeRunnable implements Runnable {
  private String name;

  public SomeRunnable(String name) {
    this.name = name;
  }

  public void run() {
    System.out.println("I am runnable " + this.name);
  }
}
```

The collection contains a mix of classes that implement `Runnable`. Since `Thread` implements `Runnable`, these can be conveniently added to the list. However, when it comes time to massively start all of the Threads, the elements within the list that are not subclasses of `Thread` may not have a "start" method. To ensure, then, that you have only `Threads`, the `InstanceSelector` is used to isolate those into their own collection, which is then supplied to the `MethodExecutor`.

Property Selectors and Iterators

It is very common to loop through a collection of beans to obtain the value of a property and then use that value for something. Wouldn't it be convenient to automate this with the simplicity of the DEF? Well, it is! Let's take a look at a couple of other classes that can help in a number of ways.

CollectionPropertySelector

The first class, the `CollectionPropertySelector` enables you to get all the values of an expression against a collection of beans. Everything you can do with an expression as described in the `BeanService` above can be used here. To use the class, simply instantiate it with a collection, and then use the "`get`" method to obtain a list of a named property.

For our example, we'll keep things simple, but will also show how cool it can be:

```
import java.util.ArrayList;

import com.jmorgan.util.collection.CollectionPropertySelector;

public class CollectionPropertySelectorExample {
  public static void main(String[] args) {
    ArrayList<String> collection = new ArrayList<String>();

    for (int i = 0; i < 25; i++) {
      collection.add("STRING" + i);
    }

    CollectionPropertySelector<String> ps =
           new CollectionPropertySelector<String>(collection);

    ArrayList<Integer> lengthList =
           (ArrayList<Integer>) ps.get("length", Integer.class);

    ArrayList<String> lcList =
           (ArrayList<String>) ps.get("toLowerCase", String.class);

    for (int i = 0; i < 25; i++) {
      System.out.printf("The length of String \"%s\" is %d " +
                        "with lower case of \"%s\"\n",
                       collection.get(i), lengthList.get(i), lcList.get(i));
    }
  }
}
```

PropertyIterator

Another class having similar use is the `PropertyIterator`. Instead of iterating through a collection, this iterator processes through the properties of the elements of a collection. Again, the property is actually an expression compatible with `BeanService`, so everything from simple bean-compliant properties to mathematical operations can be achieved:

```
import java.util.ArrayList;

import com.jmorgan.util.collection.PropertyIterator;

public class PropertyIteratorExample {
  public static void main(String[] args) {
    ArrayList<String> collection = new ArrayList<String>();

    for (int i = 0; i < 25; i++) {
      collection.add("STRING" + i);
    }

    PropertyIterator<String, Integer> lengthIterator =
          new PropertyIterator<String, Integer>(collection, "length");

    while (lengthIterator.hasNext()) {
      System.out.printf("The length is %d\n", lengthIterator.next());
    }

    PropertyIterator<String, String> lcIterator =
          new PropertyIterator<String, String>(collection, "toLowerCase");

    while (lcIterator.hasNext()) {
      System.out.printf("The lower case is %s\n", lcIterator.next());
    }
  }
}
```

Aggregators – The CollectionAggregator

When you have a collection, it is not uncommon to compute a sum, or a need to find the minimum, maximum or average of a value or values within the collection. This is the purpose of the `CollectionAggregator` class. The `CollectionAggregator` is simply a class that takes a reference to a `Collection` and one or more `AggregatorFunction`s and applies to them to the collection. An `AggregatorFunction` is simply an implementation that performs some computation against each element of the collection associated with the `CollectionAggregator`.

The `CollectionAggregator` is designed to apply one or more `AggregatorFunction`s to a collection of Java Beans, and since this class uses the DEF in full, simple or nested property aggregations are possible. To improve performance, another fundamental design of the `CollectionAggregator` is to apply as many functions as needed to the collection at one time. This means all aggregations against a collection can be applied on a single pass.

```
import java.io.File;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Map;

import com.jmorgan.math.Fraction;
import com.jmorgan.util.CountAggregator;
import com.jmorgan.util.NumericalAggregator;
import com.jmorgan.util.collection.CollectionAggregator;

public class CollectionAggregatorExample {
  public static void main(String[] args) {
    File rootDir = new File("/");
    File[] files = rootDir.listFiles();
    ArrayList<File> fileList = new ArrayList<File>(Arrays.asList(files));

    CollectionAggregator<File> aggregator =
                                new CollectionAggregator<File>(fileList);

    NumericalAggregator lengthAggregator = new NumericalAggregator();
    aggregator.addAggregatorFunction("length", lengthAggregator);

    CountAggregator<String> countAggregator = new CountAggregator<String>();
    aggregator.addAggregatorFunction("name", countAggregator);

    aggregator.aggregate();

    System.out.println("The maximum length of the files in the list is: " +
                                          lengthAggregator.getMaximum());
    System.out.println("The minimum length of the files in the list is: " +
                                          lengthAggregator.getMinimum());
    System.out.println("The average length of the files in the list is: " +
                                          lengthAggregator.getAverage());
    System.out.println("The sum of the lengths of the files in the list is: " +
                                               lengthAggregator.getSum());

    Map<String, Integer> counts = countAggregator.getCounts();
    for (String name : counts.keySet()) {
      int count = counts.get(name);
      System.out.printf("The count of occurrences of %s is: %s\n", name, count);
    }

    Map<String, Fraction> frequencies = countAggregator.getFrequencies();
    for (String name : frequencies.keySet()) {
      Fraction freq = frequencies.get(name);
      System.out.printf("The frequency of %s is: %s\n", name, freq.toString());
    }
  }
}
```

A quick note about the included `Fraction` class. Though it is not a formal part of the DEF, I developed this class to more accurately represent certain numbers, such as the frequencies you see displayed in the above sample application. Since you have access to it, though, it is only fair to see an example. `Fraction`s can be converted to decimals, and can convert decimals to fractions. We can also perform simple mathematical operations on `Fraction`s:

```
import com.jmorgan.math.Fraction;
import com.jmorgan.math.FractionalMath;

public class FractionExample {
   public static void main(String[] args) {
      Fraction oneThird = new Fraction(1, 3);
      Fraction twoSevenths = new Fraction(2, 7);

      System.out.printf("%s + %s = %s\n",
                        oneThird.toString(),
                        twoSevenths.toString(),
                        FractionalMath.add(oneThird, twoSevenths));

      System.out.printf("%s - %s = %s\n",
                        oneThird.toString(),
                        twoSevenths.toString(),
                        FractionalMath.subtract(oneThird, twoSevenths));

      System.out.printf("%s * %s = %s\n",
                        oneThird.toString(),
                        twoSevenths.toString(),
                        FractionalMath.multiply(oneThird, twoSevenths));

      System.out.printf("%s / %s = %s\n",
                        oneThird.toString(),
                        twoSevenths.toString(),
                        FractionalMath.divide(oneThird, twoSevenths));

      double value = 15.35;
      Fraction fromDecimal = new Fraction(value);
      System.out.printf("%2.2f to a fraction is: %s\n",
                       value,
                       fromDecimal.toString());

      Fraction toDecimal = new Fraction(32, 3, 5);
      System.out.printf("%s to a decimal is: %2.2f\n",
                       toDecimal.toString(),
                       toDecimal.toDecimal());
   }
}
```

Returning to the point at hand, since `AggregatorFunction` is an interface, it is easy to create custom functions to do whatever you need. When creating your own `AggregatorFunction`, there are only two methods of the interface you need to write.

```
public void start();
```

This method is invoked from the `CollectionAggregator` to tell the `AggregatorFunction` implementation that it is about to begin its pass over the collection. This enables the implementation to initialize its state, thus preventing the need to have a new aggregation function instance on each pass.

That is, suppose there is a UI displaying statistics about some data you have contained within a collection that is being dynamically modified with a thread. In this case, you'd only want one aggregator instance and at certain timed intervals, you'd want to display the results of the aggregation. You certainly don't want to create new aggregators and aggregation instances on each pass.

The other method of the `AggregatorFunction` interface is:

```
public void aggregate(T value);
```

This is the work-horse of the `AggregatorFunction`. It gets a reference to each value within the collection so it can provide the service of the implementation. For example, here are the `start` and `aggregate` methods of the `NumericalAggregator`:

```
	public void start() {
		this.sum = 0D;
		this.minimum = Double.MAX_VALUE;
		this.maximum = Double.MIN_VALUE;
		this.count = 0L;
	}

	public void aggregate(Number value) {
		double number = value.doubleValue();

		this.sum += number;

		if (number < this.minimum) this.minimum = number;
		if (number > this.maximum) this.maximum = number;

		this.count++;
	}
```

Pair

A quick explanation before the we discuss the next class, because this one is referenced and used. The `Pair` class is a value object of a pair of any two things. You likely have something like this in your arsenal of convenient things, and this framework uses mine.

In this case, the `Pair` class is also a true Java bean, with proper accessors and mutators, `equals`, `hashCode`, `compareTo` and `toString`. It is very complete and convenient. Though no requirement to use this one anywhere else, when it is used within the DEF, it is the class you'll need.

CollectionUtility

It is somewhat strange that after all of this, we'd have a utility class for collections. The utility methods within the class are quite remarkable at times.

There are many ways to obtain a differential of two collections, and in the vast majority of cases, it is just a matter of removing all elements from one collection that are contained within another, provided the `hashCode` and `equals` methods of the contained elements are properly coded, and all the elements are of the same type and have similar state.

I ran into a scenario, though, where I had two collections with dissimilar state where all would have been considered unequal and their hash codes would have been different. In this particular case, I needed to use only a few of the properties, not all of them, to determine the difference.

So, I turned to the DEF as more of a framework to create what might be the most flexible reusable diff tool on the planet! Here are the signatures of the `getDiff` methods of the `CollectionUtility`:

```
public static <T1, T2> Collection<Pair<T1, T2>> getDiff(Collection<T1> collection1,
Collection<T2> collection2, String... compareProperties)
```

```
public static <T1, T2> Collection<Pair<T1, T2>> getDiff(Collection<T1> collection1,
Collection<T2> collection2, Collection<String> compareProperties)
```

OK... a lot to absorb, but, if you look closely, you'll see the methods are static. You'll also notice the seemingly over-use of generics, but wait before you judge. The method returns a collection of `Pair` objects (The `Pair` class is described above). Each pair represents a representation of the differential between the elements within the collections.

The parameters to the methods are a collection of one kind of “thing”, a collection of another kind of “thing”, and a set of properties within those “things” to use for defining the differential. The two collections need not actually be collections of two different kinds of things, but the methods allow it.

Coming back to the return value of the methods, you'll notice the there are two types within the collection of `Pair` objects returned, those types matching, respectively, the parameterized types of the collections provided in the parameters. Some quick notes. If either collection is `null` or the properties to compare them by is `null`, these methods return `null`. If both collections are empty, an empty collection is returned. If one collection is empty, a collection is returned containing all of the elements of the non-empty collection paired with `null`. If it is the first collection that is empty, then all the elements of the second collection are returned in the second value of the `Pair` elements, with all the first values being `null`. If it is the second collection that is empty, then all the elements of the first collection are returned in the first value of the `Pair` elements, with all the second values being `null`.

Now, before you run away asking, “What in the world would I need this for?”, consider you have two collections, one of, say, a Customer, the other of Employees. You want to know which employees are customers, or vice-versa. You want a resulting list of all of them, but to quickly know which ones match. Furthermore, you likely cannot simply rely upon `equals` or `hashCode` to determine their differences (or similarities). You might can, however, compare their names and addresses.

Though not so quite as sexy an example, this one makes a pretty good case:

```
import java.util.ArrayList;
import java.util.Collection;

import com.jmorgan.util.Pair;
import com.jmorgan.util.collection.CollectionUtility;

public class CollectionUtilityExample {
  public static void main(String[] args) {
    ArrayList<Person> people = new ArrayList<>();
    ArrayList<Employee> employees = new ArrayList<>();

    String[] firstNames = { "Ben", "Jerry", "Bob", "Mary", "Alice", "Carla" };
    String[] lastNames = { "Bonn", "Jones", "Bright", "Mann", "Axel", "Cross" };

    for (int i = 0; i < firstNames.length; i++) {
      if (i != 3) people.add(new Person(firstNames[i], lastNames[i]));
      if (i != 1) employees.add(new Employee(i, firstNames[i], lastNames[i]));
    }

    Collection<Pair<Person, Employee>> diff =
      CollectionUtility.getDiff(people, employees, "firstName", "lastName");

    for (Pair<Person, Employee> e : diff) {
      System.out.println(e);
    }
  }

  static class Person {
    String firstName;
    String lastName;

    Person(String firstName, String lastName) {
      super();
      this.setFirstName(firstName);
      this.setLastName(lastName);
    }

    public String getFirstName() {
      return this.firstName;
    }

    public void setFirstName(String firstName) {
      this.firstName = firstName;
    }

    public String getLastName() {
      return this.lastName;
    }

    public void setLastName(String lastName) {
      this.lastName = lastName;
    }

    public String toString() {
      return this.firstName + " " + this.lastName;
    }
  }

  static class Employee extends Person {
    int employeeID;

    Employee(int employeeID, String firstName, String lastName) {
      super(firstName, lastName);
      this.setEmployeeID(employeeID);
    }

    public int getEmployeeID() {
      return this.employeeID;
    }

    public void setEmployeeID(int employeeID) {
      this.employeeID = employeeID;
    }

    public String toString() {
      return "#" + this.employeeID + " - " + super.toString();
    }
  }
}
```

As you can see with this example, we have two simple classes (more for brevity than anything else), a `Person` and an `Employee`. In this case, the `Employee` extends `Person` (again for brevity). I didn't go through the process of creating a proper `equals` and `hashCode` on purpose, because I wanted to prove this will work without them. This is not to say your classes shouldn't have proper `equals` and `hashCode`, but you may not always have control over the content or development of a class.

The setup simply loads the two collections with a variety of the information. The cool stuff starts when we call the `CollectionUtility.getDiff` method. Note that what we get back is:

```
(Alice Axel, #4 - Alice Axel)
(Ben Bonn, #0 - Ben Bonn)
(Bob Bright, #2 - Bob Bright)
(Carla Cross, #5 - Carla Cross)
(Jerry Jones, null)
(null, #3 - Mary Mann)
```

Which displays a full differential of people to employees and where the two differ. As you can see, Jerry Jones is not an employee, and Mary Mann isn't in the person collection. What's more important is, we decided not to compare by the full-blown class' contents, but arbitrary properties within them.

Another even more powerful thing, is that the `getDiff` method uses the `BeanService` to extract those properties. This means any expression `BeanService` supports is also supported as a compare property. So, if the `Person` and `Employee` classes above had something very complex within them such as an `Address`, the compare properties could have included things like "`address.city`" and "`address.state.abbreviation`"!

Summary

Take a deep breath. I know that was a long chapter, but, it shows you the power of the DEF, and, by now, your creative juices should really be flowing.

Chapter 5 – Additional Beans Classes

Index

Though not a formal part of the DEF, the `Index` class is like a `HashMap`, somewhat similar to `TreeMap`, allows more than one entry per key, and it is very easy and intuitive to iterate through the entries.

```
import com.jmorgan.util.Index;

public class IndexExample {
   public static void main(String[] args) {
      Index<String, String> nameIndex = new Index<String, String>();

      nameIndex.put("Smith", "Bob");
      nameIndex.put("Smith", "Mary");
      nameIndex.put("Smith", "Cindy");
      nameIndex.put("Jones", "Jane");
      nameIndex.put("Jones", "Clare");
      nameIndex.put("Jones", "Henry");

      for (String lastName : nameIndex.getKeys()) {
         for (String firstName : nameIndex.get(lastName)) {
            System.out.printf("%s, %s\n", lastName, firstName);
         }
      }
   }
}
```

BeanIndexer

The `BeanIndexer`, which is a formal part of the DEF, will help you to create an index of a collection of beans. This is very useful when needing to repeatedly process a collection of beans related to a given value, and is much faster than repeatedly using a `PropertyValueSelector` against a given property. Multiple indexes can be obtained on a single collection with a single instance of a `BeanIndexer`:

```java
import java.util.ArrayList;
import java.util.Calendar;
import java.util.Random;

import com.jmorgan.beans.util.BeanIndexer;
import com.jmorgan.util.Index;

public class BeanIndexerExample {
  private static Random r = new Random(System.currentTimeMillis());

  public static void main(String[] args) {
    ArrayList<Calendar> dateList = new ArrayList<Calendar>();

    for (int i = 0; i < 100; i++)
      dateList.add(getCalendarInstance());

    BeanIndexer<Calendar> dateIndexer = new BeanIndexer<Calendar>(dateList);
    Index<Integer, Calendar> indexByYear = dateIndexer.getIndexOf("time.year", 0);
    for (int year : indexByYear.getKeys()) {
      for (Calendar c : indexByYear.get(year)) {
        System.out.println(year + ": " + c.getTime());
      }
    }
  }

  private static Calendar getCalendarInstance() {
    Calendar c = Calendar.getInstance();

    int year = 2000 + (Math.abs(r.nextInt()) % 10);
    int month = Math.abs(r.nextInt()) % 13;
    int date = Math.abs(r.nextInt()) % 28;

    c.set(year, month, date);

    return c;
  }
}
```

Beyond the constructor which accepts the collection of beans to be indexed, the primary method of `BeanIndexer` is the `getIndexOf` method. This method takes any allowable expression defined for `MethodInvoker`. It returns an `Index` based upon the value of the expression. The second parameter into the `getIndexOf` method defines a placeholder for a `null` value, and also helps to define the parameterized type of the `Index`.

BeanPropertyLoader

The `BeanPropertyLoader` class loads a bean's properties with values provided from either individual properties or from a map of properties and their values. The use case of setting a single property value exists for completeness, and since this class uses the `BeanService` to set the value, it is really not the purpose of this class. The signature of the single property set is:

```
public void setProperty(String propertyName, Object value);
```

This method sets the given `propertyName` to the given `value`.

`BeanPropertyLoader` is best designed for setting multiple properties of a bean to the values associated within a map. The method for setting multiple properties all at once is:

```
public void setProperties(Map<String, Object> properties);
```

Where the keys of the given mapped `properties` are the names of the properties to set to their associated values.

The other classes defined within this chapter are pre-defined maps for use with this class, and supplies a set of use cases for which this class is designed.

BeanPropertyMap

`BeanPropertyMap` is an abstract class providing the basic mechanism for mapping values from a source into a bean. The source can be just about anything, such as a properties file, and XML file, JSON data, or whatever. Several subclasses exist to handle mapping for you, but you can also create your own `BeanPropertyMap` subclass.

Essentially, `BeanPropertyMap` takes a bean and maps arbitrary values to the bean's properties. Several built-in control mechanisms are available to subclasses. Implementations can control which properties should be mapped into the bean by use of the `allowMap` method. Properties from the map can be altered to conform to the actual bean property name via the `getPropertyNameFor` method. Finally, values having incompatible types can be converted via `getValueFor`.

The map is obtained within subclasses via the `getMap` method, which returns the map of the properties and values for the given implementation. For example, a class mapping a set of values from a database table to a bean might override this method to read the data from a row in a table defining the map from the names of the columns and their values.

The "workhorse" method is the `loadBean` method, which iterates through the keys of the map from the `getMap` method. For each key, it invokes `allowMap` to determine if the map is allowed for the implementation. If not allowed, the key is passed by. If allowed, it takes the value from the map obtained from the `getMap` method, and then invokes the `getPropertyNameFor` method to get the actual property name in the bean relative to the key of the map. It then invokes the `getValueFor` the key and the value obtained from the `getMap` method to get the actual value for the bean.

The property name returned from `getPropertyNameFor` and the value returned from the `getValueFor` method use used to create a new map that it then feeds into a `BeanPropertyLoader` instance to load those properties into the bean.

The general design of this class and the way it functions is to simply instantiate one of its subclasses passing it the appropriate object and the bean. Once the constructor is complete, the bean's properties, or at least the ones allowed to be set and/or those present, will be set. Therefore, the general development pattern for uses of these classes is:

```
SomeBean bean = new SomeBean();
new SomePropertyMap(ThingContainingDataToMap, bean);

// bean data is set
```

PropertiesBeanMap

This class maps the values contained within a properties file into a bean. Since all properties from a properties file are strings, subclasses should be sure to convert property values to the proper type for the bean. Of course, another strategy could be to overload non-string property mutators of the bean to allow a String as a parameter and then use that method to handle the conversion. Obviously, then, if the type of a property is a String, no special coding is required.

If the `PropertiesBeanMap` instance is to handle the data conversions, and unless the bean's properties are all strings, the class should certainly override `getValueFor`, and potentially `allowMap`. The need to override `allowMap` can be in the case where the properties within the properties file may belong to more than one bean, in which case, either multiple instances of `PropertiesBeanMap` may be used, or a single `PropertiesBeanMap` instance with `allowMap` and `getValueFor` testing the bean type and providing logic to allow or disallow, and to convert under the proper conditions.

`PropertiesBeanMap` is used by providing it with either a file name or a file, and a bean to which the file's properties will be mapped. The constructors are:

```
public PropertiesBeanMap(String propertiesFileName, ApplicationProperties bean);
public PropertiesBeanMap(File propertiesFile, ApplicationProperties bean);
```

So, given the name of a properties file, the properties are mapped to a bean something like this:

```
SomeBean bean = new SomeBean();
new PropertiesBeanMap("propertiesFileName.properties", bean);
```

CookieBeanMap

For server-side applications, it is not uncommon to receive cookies set by the application. It is desirable to always work with properties of a properly developed bean rather than to work with the cookies directly, most especially if the values within the cookies need to be passed along in a POJO based structured application.

The `CookieBeanMap`, like the `PropertiesBeanMap`, receives its name-value pairs as Strings, and so should be structured to handle conversions where necessary. Again, like `PropertiesBeanMap`, the

`CookieBeanMap` may need to override `allowMap` and `getValueFor`, and again, can be structured to map to multiple beans, or have multiple instances to map to multiple beans.

`CookieBeanMap` works by providing it with an array of cookies and an instance of a bean through a constructor:

```
public CookieBeanMap(Cookie[] cookies, T bean);
```

When there is trouble, having the ability to debug what is going on is helpful. Debugging can be turned on by use of the overloaded constructor:

```
public CookieBeanMap(Cookie[] cookies, T bean, boolean debug);
```

Use of this constructor is almost always by passing `true`, but it is up the style of the programmer.

InitParameterBeanMap

Within server-side applications, servlets may have initialization parameters. It is desirable to load these parameters into a bean containing the properly typed values of the parameters for later reference within the servlet. Like the previous two mapping classes, init parameters are strings, and must handled accordingly.

The `InitParameterBeanMap` takes a reference to a servlet and a bean, and optionally a debug flag:

```
public InitParameterBeanMap(HttpServlet servlet, T bean);
public InitParameterBeanMap(HttpServlet servlet, T bean, boolean debug)
```

HttpRequestParameterBeanMap

Maps data from a HTTP servlet request to a bean, providing the necessary data conversions. Subclasses will need to override the `getValueFor` method to return the proper value for the serviced bean if the intended value for a given bean is not a `String`. Subclasses may also need to override the `allowMap` method to prevent unwanted attempts at mapping, such as when you have a confirmation field, hidden field, or other form data not to be mapped directly to the bean.

This class needs a reference to a `HttpServletRequest` and a bean, and, like the other server-side bean mapping classes, allows setting a debug flag:

```
public HttpRequestParameterBeanMap(HttpServletRequest request, T bean);
public HttpRequestParameterBeanMap(HttpServletRequest request, T bean, boolean
debug);
```

Summary

In this chapter, we saw how we can use the `Index` and `BeanIndexer` to index collections of beans in a number of ways. We also took a look at a utility class, `BeanPropertyLoader`, which is ideally used by subclasses of `BeanPropertyMap` to map some kind of input to a bean.

The subclasses of `BeanPropertyMap` include useful classes such as `PropertiesBeanMap`, `CookieBeanMap`, `InitParameterBeanMap` and `HttpRequestParameterBeanMap`, each of which makes taking common inputs and getting their data into beans easy, intuitive and usually just a couple of lines of code at most.

Chapter 6 – Event Delegates

Introduction

The designers of Java knew that the best mechanism for event handling was via a delegation model. This model serves us very well in many cases, but after years of implementing interfaces and writing many stubs, I began wishing for a better way. This was especially true in UI programming, because, even with the delegation model, keeping things loosely coupled continued to be a challenge. I always thought it strange that to keep classes A and B loosely coupled, it requires tight coupling to a class or interface C. With the DEF, there is no coupling unless you absolutely need it, and you absolutely need it, well, never if you don't want it.

In many cases of event handling, I found my code implementing an interface only to invoke another method when the event occurred. The most extreme case is for `ActionEventListener`s, which many times just invokes some method somewhere when a UI button, menu item or link is pressed. For example, think of a "Save" button. Ordinarily, you would implement an interface and write an entire method just to invoke the "saveData" method on some controller. With the DEF, you can wire the event directly to the controller, and no interface need be implemented at all!

The design of the DEF's event handling extensions is based upon the concept of "event invokers". That is, they solve the problem of, "when this happens over here, do that over there". These classes implement the appropriate interfaces, register themselves as the event handlers for an event producer, and target a method on an object when the event occurs. Think of them as "delegating delegates". All event invokers work through dependency injection, most needing just to be instantiated and they're done.

Understand, though, that the DEF is not designed to replace the delegation event model provided in Java. The DEF is designed to enhance it where it makes most sense to do so. The general rule of thumb is this: If your code needs the details of the event object, implement the interface. If not, then use the DEF. However, with that said, all event invokers contain a reference to the event object. You will also find within these examples great cases of using the `MethodInvoker` as a function pointer for dynamically obtaining values of objects and keeping things completely uncoupled.

One thing to know, and UI programmers will absolutely love this feature, is that the event delegates described herein handle their invocations asynchronously by default, but this can be changed so the events occur synchronously.

AbstractEventInvoker

The underlying abstract class for all of the event delegates within the DEF is the `AbstractEventInvoker`. This class sets up the fundamental behavior and features shared by all of the concrete event invokers. It also contains the key ingredients to enable you to define your own event invocation delegates. Its two constructors set up everything for the dependency injection in a generic and straightforward manner:

```
public AbstractEventInvoker(Object eventProducer, Object target,
                            String methodName);
```

This constructor, like all you will see within these classes, takes a reference to an event producer, a reference to a target object, and the name of a method belonging to the target object. Semantically, when the event occurs on the event producer, invoke the named method on the given target.

```
public AbstractEventInvoker(Object eventProducer, Object target,
                            String methodName, Object...arguments);
```

This constructor is a simple overload to add arguments that will be passed to the named method of the given target. There is one very powerful feature of this you don't want to forget that is fundamental to the DEF. Any of the provided arguments can be instances of a `MethodInvoker`, which effectively creates function pointers that are invoked so their return values are provided as the arguments to the target object's method at the time the event occurs.

For Java bean compliance, since this class maintains references to the event producer, the event, the target object, the name of the method to invoke, and optionally the arguments to the method, you have the appropriate accessors and mutators:

```
public Object getEventProducer();
public T getEvent();
public Object getTarget();
public String getMethodName();
public Object[] getArguments();
```

and:

```
public void setEventProducer(Object eventProducer);
public void setEvent(T event);
public void setTarget(Object target);
public void setMethodName(String methodName);
public void setArguments(Object...arguments);
```

As an abstract super class of all the event invokers, a key method for subclasses to use so the dependency injection works as designed is the `setListenerAddRemoveMethodNames` method. Its full signature is:

```
protected void setListenerAddRemoveMethodNames(String listenerAddMethodName,
                                               String listenerRemoveMethodName);
```

This method takes the names of the methods to use to inject the invoker instance as the event listener. Subclasses will use this method to define these within their constructors based upon the event interface they implement.

For example, the `ActionEventInvoker` class (see below) implements the `ActionListener` interface. Therefore, within its constructors, it has this line of code:

```
this.setListenerAddRemoveMethodNames("addActionListener", "removeActionListener");
```

The other key method defined within this class is the `invoke` method.

```
protected void invoke();
```

This method is called by subclasses within the event handler to delegate the call to the named method on the target. Again, extending from the example of the behavior of `ActionEventInvoker`, its `actionPerformed` method is exactly this:

```
public void actionPerformed(ActionEvent e) {
      this.setEvent(e);
      this.invoke();
}
```

As mentioned before, the default behavior of the `invoke` method of this class is to invoke the named method of the target object asynchronously when the event occurs. This behavior can be controlled by setting the `invokeSynchronously` flag:

```
public void setInvokeSynchronously(boolean invokeSynchronously);
```

Where passing `true` means to call the named target value synchronously.

Let now take a look at the implementations that exist within the DEF.

PropertyEventInvoker

Java beans and other classes that fire `PropertyChangeEvent`s to `PropertyChangeListener`s are very common. In most cases, interested classes may implement the `PropertyChangeListener` interface, register to the bean, and then handle the event when a property of interest changes. This pattern is repeated over and over. With the `PropertyChangeInvoker` class, the pattern can be implemented in a much more straightforward manner.

The two constructors are:

```
public PropertyChangeInvoker(Object eventProducer, Object target,
                             String methodName);
public PropertyChangeInvoker(Object eventProducer, Object target,
                             String methodName, Object... arguments);
```

Where `eventProducer` is an instance of a class containing the `addPropertyChangeListener` and `removePropertyChangeListener` methods. The `target` is an instance of a class containing a method defined by `methodName`, and if the method identified by `methodName` requires `arguments`, the 2nd constructor can receive those arguments, which can be surrogates defined by instances of `MethodInvoker`.

Once constructed, every time the `eventProducer` fires a property change event, `methodName` on the given `target` is invoked. This may not be ideal, since the `target` may not be interested in every property change event that may be fired by the `eventProducer`. Therefore, `PropertyChangeInvoker` allows you to define a matching pattern to limit the events sent to the `target`:

```
public void setPropertyMatchPattern(String propertyMatchPattern);
```

The property match pattern can be any legal regular expression, and the default value is ".*".

If your target is not receiving change notifications as expected, or maybe it is receiving to many notifications, it might be necessary to "see" the names of the properties that are changing. This can be done by setting the `showPropertyNames` flag via:

```
public void setShowPropertyNames(boolean showPropertyNames);
```

As you might expect, passing `true` to this method will cause the `PropertyChangeInvoker` to show the names of the properties changing on the `eventProducer`.

Sample usage of the class might look something like this:

```
import com.jmorgan.beans.util.PropertyChangeInvoker;

public class PropertyEventInvokerExample {
   private SomeBean someBean;

   public PropertyEventInvokerExample() {
      this.someBean = new SomeBean();
      new PropertyChangeInvoker(someBean, this, "beanStateChanged");
   }

   public void beanStateChanged() {
      System.out.println("Bean State Changed");
   }

   // Other code that changes properties of the bean... in which case the
   // beanStateChanged method will be called.

   public static void main(String[] args) {
      new PropertyEventInvokerExample();
   }
}
```

PropertyBinder

There are times when handling property changes in one bean means changing properties in another. Ordinarily, this might mean having one instance listen to another directly, or developing a "controller" to handle the decoupling of the two beans. `PropertyBinder` is just such a controller, designed to generically handle synchronization of bean property values while keeping those beans (or classes) completely decoupled.

The basic implementation of the use of the class is as simple as this:

```java
import com.jmorgan.beans.util.PropertyBinder;

public class PropertyBinderExample {
   public PropertyBinderExample() {
      SomeBean someBean = new SomeBean();
      SomeOtherBean someOtherBean = new SomeOtherBean();

      new PropertyBinder(someBean, "someProperty",
                         someOtherBean, "targetProperty");

      // The rest of the application that may change properties
      // on someBean, in which case someOtherBean's property will
      // be synchronized.
   }
}
```

The semantics of the constructor in the example is essentially when `someProperty` of `someBean` changes, then set `targetProperty` of `someOtherBean` to that new value.

Not all properties will have the same data type, and so something must be provided to give the binder a means to convert the data type properly. Implementations of the `DataTypeConverter` interface (see below), is used for handling the conversion. All of this can be accomplished with a constructor:

```java
import com.jmorgan.beans.util.PropertyBinder;
import com.jmorgan.beans.util.StringToDoubleConverter;

public class PropertyBinderExample2 {
   public PropertyBinderExample() {
      SomeBean someBean = new SomeBean();
      SomeOtherBean someOtherBean = new SomeOtherBean();

      new PropertyBinder(someBean, "someProperty",
                         someOtherBean, "targetProperty",
                         new StringToDoubleConverter());
   }
}
```

In the above example, `someProperty` of `someBean` is a String, while `targetProperty` of `someOtherBean` is a double. Therefore, when the `PropertyBinder` notices the property change on the source bean, it hands off the value to the data converter, which converts the String to a double. The `PropertyBinder` then sets the target property to the converted value.

A single instance of the `PropertyBinder` can be used for many bindings:

```
import com.jmorgan.beans.util.PropertyBinder;
import com.jmorgan.beans.util.StringToDateConverter;

public class PropertyBinderExample3 {
   public PropertyBinderExample() {
      SomePOJOBean bean = new SomePOJOBean();
      SomeVisualComponent component1 = new SomeVisualComponent();
      SomeVisualComponent component2 = new SomeVisualComponent();
      SomeVisualComponent component3 = new SomeVisualComponent();
      StringToDateConverter stdc = new StringToDateConverter();

      PropertyBinder binder = new PropertyBinder();
      binder.addBindingMap(component1, "value1", bean, "property1");
      binder.addBindingMap(component2, "value2", bean, "property2");
      binder.addBindingMap(component3, "value3", bean, "property3", stdc);
   }
}
```

As you can see, a single `PropertyBinder` instance works via a collection of property binding maps containing the instructions for what property changes to look for from the source beans, and what properties to set on the target beans, and any data type conversions needing to be made.

DataTypeConverter

The data type converter interface and its implementations can be used for any data type conversion. The `PropertyBinder` has been designed to use the `DataTypeConverter` for handling conversions for data types while binding one bean's property to another. Though this document does not detail examples of all of the `DataTypeConverter` implementations, know what implementations exist and that you can create any number of them you need.

Implementations within the DEF include a series of classes named with the pattern: `NumberTo{`*`AnyOtherNumber`*`}Converter`. Another set of implementations have the pattern: `StringTo{`*`Integer|Long|Float|Double`*`}Converter`. These implementations do as they are named. Other String based converters are: `StringTo{`*`Character|Date|StringArrayList`*`}Converter`.

Using a `DataTypeConverter` is as simple as knowing the conversion needing to be completed, and, in the best practices of general object orientation, are simple, do-one-thing-and-one-thing-well implementations:

```
import com.jmorgan.beans.util.StringToDoubleConverter;

public class DataTypeConverterExample {
   public static void main(String[] args) {
      String someStringNumber = "123.45";

      StringToDoubleConverter s2d = new StringToDoubleConverter();
      double number = s2d.convert(someStringNumber);

      System.out.println(someStringNumber + " converts to " + number);

      s2d = new StringToDoubleConverter(-1D);
      number = s2d.convert(null);

      System.out.println("null converts to " + number);
   }
}
```

As you can see from the above example, the concrete `StringToDoubleConverter` implementation converts strings to double values, which makes it a great service to `PropertyBinder` in UI applications. All implementations of `DataTypeConverter` within the DEF allow you to provide a placeholder for null values.

NamedPropertyBinder

Very, very similar to `PropertyBinder` is the `NamedPropertyBinder`. The difference is that the target method of the `NamedPropertyBinder` receives a name with the property value:

```
import com.jmorgan.beans.util.NamedPropertyBinder;
import com.jmorgan.beans.util.StringToDateConverter;

public class NamedPropertyValueExample {
   public static void main(String[] args) {
       SomePOJOBean bean = new SomePOJOBean();
       SomeVisualComponent component1 = new SomeVisualComponent();
       SomeVisualComponent component2 = new SomeVisualComponent();
       SomeVisualComponent component3 = new SomeVisualComponent();
       StringToDateConverter stdc = new StringToDateConverter();

       NamedPropertyBinder binder = new NamedPropertyBinder();
       binder.addBindingMap(component1, "value1", bean, "property1", "setProperty");
       binder.addBindingMap(component2, "value2", bean, "property2", "setProperty");
       binder.addBindingMap(component3, "value3", bean, "property3", "setProperty",
                            stdc);
   }
}
```

In the above example, the `SomePOJOBean` class contains a method, `setProperty`, that is invoked something like this when using the method directly:

```
bean.setProperty("property1", value);
```

Like `PropertyBinder`, many bindings can be created between any set of sources to any set of targets with a single instance. Also like `PropertyBinder`, when assistance is needed to make data type conversions during property binding, the `DataTypeConverter` of the appropriate implementation can be used.

Summary

Chapter 7 – UI Event Delegates

Introduction

All of the event delegates within this chapter are designed for UI applications. Many of the delegates will work in any UI application, applets and stand-alone applications alike. All of the UI delegates extend from `AbstractEventInvoker`, but are purpose built delegates for a particular event producer.

Another feature of all of the UI delegates is they register themselves to the event producer as the listener for the particular event. When the producer fires the event, these delegates are notified and, in turn, invoke a target method. Like the rest of the DEF and like the event delegates mentioned above, the event invoker can send parameters to the target method, and, if a `MethodInvoker` is registered as a parameter, it acts as a function pointer that is invoked at the time the event is consumed, passing the return value as the parameter for the target method.

Please note that these delegates consume and maintain a reference to the event. Though these classes are not designed to completely replace the standard Java UI event handling design patterns, when you don't necessarily need the event, as is the case in many situations, these classes become very handy, as they generally require a single line of code, no interface implementations, and no empty stubs because the app only needs to react.

Another similarity with the above event invokers is that when the event occurs, the target method is invoked asynchronously. Sometimes this is not ideal, and so all of these classes support the basic feature within the `AbstractEventInvoker` to indicate that the event and its target method should be handled synchronously.

The example code herein utilize a number of Swing extensions I have developed over the years. It is not the goal of this manual or the DEF to support these extensions or detail how they work, but they enable me to present these examples in a very flexible and quick way without all the fluff that will distract from the intent of the examples. These classes are also included in the `jmorgan.jar` file, so these should work just fine.

ActionEventInvoker

One of the most handled events in UI programming, and in other Java programming, is the `ActionEvent`. One of the very early developments of the DEF resulted from my observation that virtually every time I implemented `ActionListener`, I found that the `actionPerformed` method almost never needed the incoming `ActionEvent`, and all the method did was to invoke some other method on some other class. This almost always required some degree of coupling between two classes, generally breaking the MVC pattern.

The `ActionEventInvoker` registers as an action listener to an action event producer and takes a reference to an object and a method name or reference, and an optional set of arguments. Here's the typical setup:

```
import java.awt.Container;

import javax.swing.JButton;
import javax.swing.JPanel;

import com.jmorgan.swing.JMFrame;
import com.jmorgan.swing.event.ActionEventInvoker;

public class ActionEventInvokerExample extends JMFrame {
   public ActionEventInvokerExample() {
      super("ActionEventInvokerExample");
   }

   public Container buildGUI() {
      JPanel ui = new JPanel();

      JButton button = new JButton("Click Me");

      new ActionEventInvoker(button, button, "setText", "I've been clicked!");

      ui.add(button);

      return ui;
   }

   public static void main(String[] args) {
      new ActionEventInvokerExample();
   }
}
```

In the example, a `JButton` is created and added to a simple `JPanel`. The key things to notice are how nothing explicitly implements `ActionListener`, there is no call to `addActionListener`, and there is no `actionPerformed`. The `ActionEventInvoker`'s constructor is taking, in left to right order, the `ActionEvent` producer, the target object, the target object's method name, and an argument to pass to that method. Since `JButton` instances have a `setText` method that takes a String parameter, we provide a String to pass to that method when the button is clicked.

So the sequence of events is this:

1. The button is clicked and it fires an `ActionEvent` to all registered listeners
2. The `ActionEventInvoker`, which registered as an `ActionListener` to the button when it was constructed, receives notification of the event.
3. The `ActionEventInvoker` then invokes the `setText` method of the `JButton` passing the String `“I've been clicked!”`

Before moving along to the other delegates, let's examine another more exotic example, where the string passed as the parameter to the method is dynamically obtained at the time the event occurs:

```
import java.awt.Container;

import javax.swing.JButton;
import javax.swing.JPanel;

import com.jmorgan.lang.MethodInvoker;
import com.jmorgan.swing.JMFrame;
import com.jmorgan.swing.event.ActionEventInvoker;

public class ActionEventInvokerExample2 extends JMFrame {
   private int counter = 0;

   public ActionEventInvokerExample2() {
      super("ActionEventInvokerExample2");
   }

   public Container buildGUI() {
      JPanel ui = new JPanel();

      JButton button = new JButton("Click Me");

      MethodInvoker<String> getButtonTextInvoker =
         new MethodInvoker<String>(this, "getButtonText");

      new ActionEventInvoker(button, button, "setText", getButtonTextInvoker);

      ui.add(button);

      return ui;
   }

   private String getButtonText() {
      return "I've been clicked " + ++counter + " times!";
   }

   public static void main(String[] args) {
      new ActionEventInvokerExample2();
   }
}
```

When you run the above example, click the button several times.

There is a new method, `getButtonText`, that computes and returns a String when invoked. Notice, too, the use of `MethodInvoker` as a function pointer to this new `getButtonText` method. When the `ActionEventInvoker` is created, it set itself up to listen to the button's action events, and to target the button's `setText` method.

This is where it gets interesting. When we create the `ActionEventInvoker`, instead of passing in a String as a parameter for the `setText` method, we pass in the `MethodInvoker` as a delegate for obtaining the value of the String at the time the button is clicked. In order for this to work, though, the return value of the method associated with the `MethodInvoker` must be a String.

So the new sequence of events is this:

1. The button is clicked and it fires an `ActionEvent` to all registered listeners

2. The `ActionEventInvoker`, which registered as an `ActionListener` to the button when it was constructed, receives notification of the event.
3. The `ActionEventInvoker` then notices that the parameter to the `setText` method of the `JButton` is not a String, but an instance of `MethodInvoker`.
4. The `ActionEventInvoker` then invokes the `MethodInvoker`, which calls `getButtonText`, obtaining it's return value, which is a String.
5. The return value from the `MethodInvoker` is then passed as the parameter into the `setText` method of the `JButton` instance.

Really cool! Extremely powerful! And it only took a tiny bit more code!

We won't always get so exotic with the remaining delegates, but understand that whenever you are dynamically invoking a method with the DEF, if that method takes parameters, those can be obtained by use of the `MethodInvoker`. And, whenever setting up a `MethodInvoker`, any parameters needing to be passed to its associated method can also be delegated to a `MethodInvoker`, and so on!

As you can see, the DEF can be quite powerful especially when you consider that you can stack invoker delegates together in whatever way is necessary to assist with the processing, all with very little actual code!

ChangeEventInvoker

Change events come from a couple of UI elements; sliders and spinners most notably. Unfortunately, the `ChangeEvent` doesn't come with much information, so completely decoupling the event producer from the event handling is desired. In this situation, then, we can again turn to the DEF to help direct the result of a change event in a more useful way:

```
import java.awt.BorderLayout;
import java.awt.Container;
import java.awt.Dimension;

import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JSlider;
import javax.swing.JTextField;

import com.jmorgan.lang.MethodInvoker;
import com.jmorgan.swing.JMFrame;
import com.jmorgan.swing.event.ChangeEventInvoker;

public class ChangeEventInvokerExample extends JMFrame {
   public ChangeEventInvokerExample() {
      super("ChangeEventInvokerExample");
   }

   public Container buildGUI() {
      JPanel ui = new JPanel(new BorderLayout());

      JSlider slider = new JSlider(0, 400, 0);

      JLabel label = new JLabel("Slide the Slider");
      JPanel labelPanel = new JPanel(null, true);
      labelPanel.setPreferredSize(new Dimension(500, 50));
      label.setBounds(0, 10, 100, 20);
      labelPanel.add(label);

      MethodInvoker<Integer> getValueMethodInvoker =
         new MethodInvoker<Integer>(slider, "getValue");

      new ChangeEventInvoker(slider, label, "setLocation",
                             getValueMethodInvoker, 10);

      ui.add(labelPanel, BorderLayout.NORTH);
      ui.add(slider, BorderLayout.CENTER);

      return ui;
   }

   public static void main(String[] args) {
      new ChangeEventInvokerExample();
   }
}
```

The above example creates a UI with a slider and a label. The label is sitting in a panel using a `null` layout so it can be freely positioned. When the UI runs and the user slides the slider, the label moves horizontally. In order to move the label, we use the `setLocation` method. In this case, it is important that we use the `setLocation` method that takes the `x` and `y` individually, rather than the one that takes a `Point` instance.

In setting up the event handling, we first setup a `MethodInvoker` that will obtain the value of the slider when invoked. Then we define the event handler to use the `ChangeEventInvoker`, which registers itself as a `ChangeListener` to the slider. When the `ChangeEventInvoker` receives a change event, it invokes the `setLocation` method of the label, passing the `getValueMethodInvoker` as the parameter for `x`, and a constant `10` as the parameter for `y`.

When the DEF sees that the argument for the `x` parameter is a `MethodInvoker`, it invokes it, thus calling the `getValue` method of the slider capturing the return value, which it then passed in as the `x` value for the `setLocation` method.

ItemEventInvoker

Checkboxes and radio buttons, among other things, fire `ItemEvents`. The `ItemEvent` does contain useful information, but there are cases where we may not care so much about the contents of the event as much as we care that the event occurred. This is where the DEF comes in:

```java
import java.awt.Container;
import java.awt.Font;

import javax.swing.ButtonGroup;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JRadioButton;
import javax.swing.JSeparator;

import com.jmorgan.lang.MethodInvoker;
import com.jmorgan.swing.JMFrame;
import com.jmorgan.swing.event.ItemEventInvoker;

public class ItemEventInvokerExample extends JMFrame {
   public ItemEventInvokerExample() {
      super("ItemEventInvokerExample");
   }

   public Container buildGUI() {
      JPanel ui = new JPanel();

      JRadioButton rbBold = new JRadioButton("Bold");
      JRadioButton rbItalic = new JRadioButton("Italic");
      ButtonGroup bg = new ButtonGroup();
      bg.add(rbBold);
      bg.add(rbItalic);

      JLabel label = new JLabel("Some Label");

      Font labelFont = label.getFont();
      label.setFont(labelFont.deriveFont(Font.PLAIN));

      Font boldFont = labelFont.deriveFont(Font.BOLD);
      Font italicFont = labelFont.deriveFont(Font.ITALIC);

      new ItemEventInvoker(rbBold, label, "setFont", boldFont);
      new ItemEventInvoker(rbItalic, label, "setFont", italicFont);

      ui.add(label);
      ui.add(rbBold);
      ui.add(rbItalic);

      return ui;
   }

   public static void main(String[] args) {
      new ItemEventInvokerExample();
   }
}
```

Though not the greatest example, we can see how when item events are triggered, we can steer the result directly to a desired action using the `ItemEventInvoker`, rather than adding a lot more code in the form of interface implementations.

ListSelectionEventInvoker

The ListSelectionEventInvoker routes a notification of a ListSelectionEvent to any method in much the same way as do the previously detailed event invokers. This class is here as a matter of completeness in the DEF, but the ListSelectionEvent is useful more often than not. However, because we can utilize the mechanisms within the DEF robustly in our code, it still has quite a bit of merit.

```
import java.awt.Container;

import javax.swing.JList;
import javax.swing.JPanel;

import com.jmorgan.lang.MethodInvoker;
import com.jmorgan.swing.JMFrame;
import com.jmorgan.swing.event.ListSelectionEventInvoker;

public class ListSelectionEventInvokerExample extends JMFrame {
  public ListSelectionEventInvokerExample() {
    super("ListSelectionEventInvokerExample");
  }

  public Container buildGUI() {
    JPanel ui = new JPanel();

    JList list = new JList(new String[] { "One", "Two", "Three", "Four" });

    MethodInvoker<Object> getValueInvoker =
          new MethodInvoker<Object>(list, "getSelectedValue");

    new ListSelectionEventInvoker(list, this, "setTitle", getValueInvoker);

    ui.add(list);

    return ui;
  }

  public static void main(String[] args) {
    new ListSelectionEventInvokerExample();
  }
}
```

An important thing to know about the ListSelectionEventInvoker is that it only routes to the target method when the value is not adjusting. That is, the target method is only invoked after the list selection change completes. This prevents multiple calls to the target method. This is an implementation decision being that the target method generally does not have access to the ListSelectionEvent object to be able to know anything about the event. Again, I note, that if your program needs to know details of the event that cannot be derived by some means, you can either hold a reference to the ListSelectionEventInvoker and then use getEvent(), or use the standard event listening development pattern.

AbstractMaskedEventInvoker

The last four event invokers handle event objects that are fired based upon a single method event. That is, an `ActionEvent` is an `ActionEvent`, it doesn't matter if the CTRL key is pressed or if the mouse pointer is on the left side of a button or menu item. The same is true of the `ChangeEvent`, the `ItemEvent`, and the `ListSelectionEvent`.

Though a pretty good argument could be made that `ListSelectionEvent` is a multi-state event, being that it is fired before, during and after a list selection change, it is included here because the event handlers for these events have a single method defined within their related interfaces.

The following event invokers truly have different conditions under which they are fired, and many, many empty stubs have been written to "handle" conditions in which the program isn't the least bit interested. Measures have been taken with the out-of-the-box Java to provide "adapter" classes for these handlers, but I have found far more often than not that programs cannot effectively utilize these adapters without writing complex facade wrappers around them. This is especially true when a controller needs to handle more than one of these types of events, such as handling both `MouseEvents` and `FocusEvents`.

The event invokers for the related events that follow contain more than one method to "handle" the many kinds of events that can actually occur. This is addressed within the DEF by subclasses of the `AbstractMaskedEventInvoker`. The idea of the "mask" is to be able to clearly identify under what conditions the target method is to be invoked. The setup of these invokers is very much the same as it is for the single method event handlers, except we add the ability to build a mask that sets the rules for when the target method is called. Different invokers have different, but quite intuitive, masking values, and they can be combined to instruct the invoker to call the target method when more than one kind of event occurs.

What this means to you as a programmer is you still have no interface to implement, no more empty stubs to write, and no more convoluted anonymous or adapter code to invent to try and handle multiple cases. With all that said, however, it is not uncommon to need the event object produced from these events, such as when you need the actual location of the mouse, or the actual keys being pressed. In these cases, it is not recommended to try and use the DEF or these event invokers, but to code the event handler the standard way. However, when you do not need the actual event object, and you just want to react to the fact that the mouse was clicked, or a key was pressed, these invokers work wonders.

ComponentEventInvoker

Component event producers send events under four different conditions: when the component is shown, moved, hidden, or resized. `ComponentListener`, therefore, has four methods. What if, though, there is a part of your program that only needs to know if the component has moved, or needs to do the same thing regardless if the component is hidden or shown?

Providing a solution to that question is the goal of the `ComponentEventInvoker`. As a masked event invoker, when you construct the `ComponentEventInvoker`, you provide it a triggering mask to tell it under what conditions to actually invoke its target. The masks are:

```
COMPONENT_HIDDEN
COMPONENT_SHOWN
COMPONENT_MOVED
COMPONENT_RESIZED
```

Like all event invokers, the `ComponentEventInvoker` registers itself to its designated component and is provided a target method and an object or class reference that will be invoked when an event occurs, and allows parameters to be provided for the target method. There are no mysteries here, `MethodInvoker`s are allowed as parameter delegates, just like all previous cases.

```java
import java.awt.BorderLayout;
import java.awt.Container;
import java.awt.Dimension;

import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JPanel;

import com.jmorgan.lang.MethodInvoker;
import com.jmorgan.swing.JMFrame;
import com.jmorgan.swing.event.ComponentEventInvoker;
import com.jmorgan.swing.util.VisualComponentEditor;

public class ComponentEventInvokerExample extends JMFrame {
   public ComponentEventInvokerExample() {
      super("ComponentEventInvokerExample");
   }

   public Container buildGUI() {
      JPanel ui = new JPanel(new BorderLayout());

      JLabel label = new JLabel("Move Me with the mouse");
      label.setBounds(10, 10, 150, 20);
      JButton button = new JButton("Watch me move with the label");
      button.setBounds(10, 10, 250, 25);

      new VisualComponentEditor(label);

      MethodInvoker<Point> getLocationInvoker =
         new MethodInvoker<Point>(label, "getLocation");

      new ComponentEventInvoker(label, ComponentEventInvoker.COMPONENT_MOVED,
                                button, "setLocation", getLocationInvoker);

      JPanel labelPanel = new JPanel(null);
      labelPanel.add(label);
      labelPanel.setPreferredSize(new Dimension(10, 200));

      JPanel buttonPanel = new JPanel(null);
      buttonPanel.add(button);

      ui.add(labelPanel, BorderLayout.NORTH);
      ui.add(buttonPanel, BorderLayout.CENTER);

      ui.setPreferredSize(new Dimension(600, 400));

      return ui;
   }

   public static void main(String[] args) {
      new ComponentEventInvokerExample();
   }
}
```

The above example demonstrates how simple the `ComponentEventInvoker` makes the visual effect possible. Essentially, when the user moves the label, the button moves with it. Several elements of the DEF are employed, here, including the now famous `MethodInvoker`. Another class, the `VisualComponentEditor`, handles the visual and functional trickery to enable the user to move the label, but it is not part of the DEF, yet it is included with `jmorgan.jar`, so I'll utilize it to keep the example on topic.

We first set up a label and a button, and the bounds they will occupy within their containers. Then we setup a `VisualComponentEditor` and associate it with the label. The `MethodInvoker` references the `getLocation` method of the label, which will provide the value needed for the `setLocation` method of the button when the label is moved.

When we create the `ComponentEventInvoker`, note the mask parameter, `ComponentEventInvoker.`*`COMPONENT_MOVED`*. This tells the event invoker to only call `setLocation` on the button when the label is moved, but not when the label is hidden, shown, or resized. In this example, the label can be resized, and if you resize it to the left or upwards, it appears the resizing causes the button to move. The truth is, though, that when you resize the button to the left or upwards, the location of the label is affected, thus causing the `componentMoved` event to occur.

To demonstrate an example of combing masks, lets make it so the button remains the same bounds as the label all the time:

```
import java.awt.BorderLayout;
import java.awt.Container;
import java.awt.Dimension;

import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JPanel;

import com.jmorgan.lang.MethodInvoker;
import com.jmorgan.swing.JMFrame;
import com.jmorgan.swing.event.ComponentEventInvoker;
import com.jmorgan.swing.util.VisualComponentEditor;

public class ComponentEventInvokerExample2 extends JMFrame {
   public ComponentEventInvokerExample2() {
      super("ComponentEventInvokerExample");
   }

   public Container buildGUI() {
      JPanel ui = new JPanel(new BorderLayout());

      JLabel label = new JLabel("Move and Resize Me with the mouse");
      label.setBounds(10, 10, 250, 20);
      JButton button = new JButton("Watch me change with the label");
      button.setBounds(10, 10, 250, 25);

      new VisualComponentEditor(label);

      MethodInvoker<Rectangle> getBoundsInvoker =
         new MethodInvoker<Rectangle>(label, "getBounds");

      new ComponentEventInvoker(label,
                                ComponentEventInvoker.COMPONENT_MOVED |
                                ComponentEventInvoker.COMPONENT_RESIZED,
                                button, "setBounds", getBoundsInvoker);

      JPanel labelPanel = new JPanel(null);
      labelPanel.add(label);
      labelPanel.setPreferredSize(new Dimension(10, 200));

      JPanel buttonPanel = new JPanel(null);
      buttonPanel.add(button);

      ui.add(labelPanel, BorderLayout.NORTH);
      ui.add(buttonPanel, BorderLayout.CENTER);

      ui.setPreferredSize(new Dimension(400, 400));

      return ui;
   }

   public static void main(String[] args) {
      new ComponentEventInvokerExample2();
   }
}
```

The main differences in the above example is the method pointed to by the `MethodInvoker`, the target method of the `ComponentEventInvoker`, and the combined mask that invokes the target method if the label is resized or moved. Note the mask is built of multiple event types by using the "|" character.

ContainerEventInvoker

The `ContainerEventInvoker` responds to components being added or removed from a container. On construction, it registers itself as a `ContainerListener` to any valid container. You may need this in a dynamic environment where such a thing may occur, which is the reason for its existence. The masks for this class are:

```
    COMPONENT_ADDED
    COMPONENT_REMOVED
```

Here is an example that when a button is dynamically added to and removed from a container, a message is shown:

```
import java.awt.Container;
import java.awt.Dimension;

import javax.swing.JButton;
import javax.swing.JOptionPane;
import javax.swing.JPanel;

import com.jmorgan.lang.AsynchMethodInvoker;
import com.jmorgan.swing.JMFrame;
import com.jmorgan.swing.event.ContainerEventInvoker;

public class ContainerEventInvokerExample extends JMFrame {
   public ContainerEventInvokerExample() {
      super("ContainerEventInvokerExample");
   }

   public Container buildGUI() {
      JPanel ui = new JPanel();

      new ContainerEventInvoker(ui, ContainerEventInvoker.COMPONENT_ADDED,
                                this, "showEventMessage",
                                "Hey, something was added!  Click me quick!");

      new ContainerEventInvoker(ui, ContainerEventInvoker.COMPONENT_REMOVED,
                                this, "showEventMessage",
                                "Hey, something was removed!");

      JButton button = new JButton("Hello, I'm a button");

      new AsynchMethodInvoker<Void>(this, "addButton", button, 3000);
      new AsynchMethodInvoker<Void>(this, "removeButton", button, 8000);

      ui.setPreferredSize(new Dimension(200, 100));

      return ui;
   }

   private void addButton(JButton button) {
      this.getGUIPane().add(button);
      this.getGUIPane().validate();
```

```
    }

    private void removeButton(JButton button) {
        this.getGUIPane().remove(button);
    }

    private void showEventMessage(String message) {
        JOptionPane.showMessageDialog(this, message,
                                          "ContainerEventInvoker",
                                          JOptionPane.INFORMATION_MESSAGE);
    }

    public static void main(String[] args) {
        new ContainerEventInvokerExample();
    }
}
```

The code is pretty intuitive. A blank panel is initially created. Two `ContainerEventInvokers` are created listening for `ContainerEvents` on the panel, one for each event, an target the `showEventMessage` method of the frame. A button is created which is not immediately added to the screen, but added 3 seconds later via an `AsynchMethodInvoker` call to `addButton`. When `addButton` is eventually called, the container event fires routing through both of the `ContainerEventInvoker` instances. However, since only one of them is listening for components to be added, only one of them invokes `showEventMessage`.

A similar thing happens when 5 seconds after the button is added, it is removed. Again, both `ContainerEventInvokers` are notified, but since only one of them will invoke its target method when a component is removed, we see the correct message.

One other thing the example emphasizes. Notice the `addButton`, `removeButton`, and `showEventMessage` methods are `private`. This is allowed in run-time environments where the `SecurityManager` permits it. This would not work as an applet, because the `SecurityManager` doesn't allow it when running within a browser. Just a note that `private`, `protected` and `package` visible methods can be targets under the right circumstances.

FocusEventInvoker

`FocusEventInvoker` is as it sounds, firing target methods based upon components gaining and/or losing focus. `FocusEventInvoker` has two masks:

```
FOCUS_GAINED
FOCUS_LOST
```

By now, you're getting a clear idea, so here's a fun example:

```
import java.awt.Color;
import java.awt.Container;
import java.awt.Dimension;

import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;

import com.jmorgan.lang.DynamicProcess;
import com.jmorgan.lang.MethodInvoker;
import com.jmorgan.swing.JMFrame;
import com.jmorgan.swing.event.FocusEventInvoker;

public class FocusEventInvokerExample extends JMFrame {
   public FocusEventInvokerExample()  {
      super("FocusEventInvokerExample");
   }

   public Container buildGUI() {
      JPanel ui = new JPanel();

      JTextField firstFocus = new JTextField("First Focus");
      JTextField secondFocus = new JTextField("Second Focus");
      JLabel status = new JLabel("");
      status.setOpaque(true);

      new FocusEventInvoker(firstFocus, FocusEventInvoker.FOCUS_LOST,
                            status, "setBackground", Color.YELLOW);
      new FocusEventInvoker(secondFocus, FocusEventInvoker.FOCUS_GAINED,
                            status, "setText", "Second Field Gained Focus");

      DynamicProcess dynProcess =
         new DynamicProcess(new MethodInvoker(status, "setText",
                                               "First Field Gained Focus"),
                            new MethodInvoker(status, "setBackground", Color.WHITE));
      new FocusEventInvoker(firstFocus, FocusEventInvoker.FOCUS_GAINED,
                              dynProcess, "invoke");

      ui.add(firstFocus);
      ui.add(secondFocus);
      ui.add(status);

      ui.setPreferredSize(new Dimension(400, 200));

      return ui;
   }

   public static void main(String[] args) {
      new FocusEventInvokerExample();
   }
}
```

So, two text fields and a label to display information. The first two `FocusEventInvokers` are single mask event delegates making changes to properties of the label based upon the first text field losing focus, and the second field gaining focus. The third `FocusEventInvoker` invokes a `DynamicProcess` that changes both the text and background of the label when the first text field gains focus. You're beginning to see just how flexible and powerful the DEF can become.

HyperlinkEventInvoker

In a world of richer displays of information, it is not uncommon to display HTML even within stand-alone applications and applets. It is also nice to be able to embed links that can be used to trigger behavior within the application. There are three types of `HyperLinkEvents`, and they are masked for `HyperlinkEventInvoker` as:

```
HYPERLINK_ACTIVATED
HYPERLINK_ENTERED
HYPERLINK_EXITED
```

Like the other masked event invokers, it is normally as simple as constructing them with the appropriate mask:

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Container;

import javax.swing.BorderFactory;
import javax.swing.JEditorPane;
import javax.swing.JPanel;

import com.jmorgan.swing.JMFrame;
import com.jmorgan.swing.event.HyperlinkEventInvoker;

public class HyperlinkEventInvokerExample extends JMFrame {
   public HyperlinkEventInvokerExample() {
      super("HyperlinkEventInvokerExample");
   }

   public Container buildGUI() {
      JPanel ui = new JPanel(new BorderLayout());

      JEditorPane editorPane = new JEditorPane("text/html", this.getHTML());
      editorPane.setEditable(false);

      new HyperlinkEventInvoker(editorPane,
                                HyperlinkEventInvoker.HYPERLINK_ACTIVATED,
                                ui, "setBorder",
                       BorderFactory.createLineBorder(Color.red));
      new HyperlinkEventInvoker(editorPane,
                                HyperlinkEventInvoker.HYPERLINK_ENTERED,
                                ui, "setBorder",
                       BorderFactory.createEtchedBorder());
      new HyperlinkEventInvoker(editorPane,
                                HyperlinkEventInvoker.HYPERLINK_EXITED,
                                ui, "setBorder", BorderFactory.createEmptyBorder());

      ui.add(editorPane);

      return ui;
   }

   private String getHTML() {
      return "<html><body>" +
                "<p>This contains <a href=\"link1\">a link</a></p>" +
                "</html></body>";
   }

   public static void main(String[] args) {
      new HyperlinkEventInvokerExample();
   }
}
```

So, this is pretty cool, we can see that each state of a `HyperLinkEvent` is handled quite easily. But, what if there is more than one link and you need to react to them separately?

In addition to the mask, `HyperlinkEventInvoker` also filters its behavior based upon a URL matching pattern. The default pattern is "`.*`", which matches everything, but you can change this via the `setUrlMatchPattern` method:

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Container;

import javax.swing.BorderFactory;
import javax.swing.JEditorPane;
import javax.swing.JPanel;

import com.jmorgan.swing.JMFrame;
import com.jmorgan.swing.event.HyperlinkEventInvoker;

public class HyperlinkEventInvokerExample2 extends JMFrame {
   public HyperlinkEventInvokerExample2() {
      super("HyperlinkEventInvokerExample");
   }

   public Container buildGUI() {
      JPanel ui = new JPanel(new BorderLayout());

      JEditorPane editorPane = new JEditorPane("text/html", this.getHTML());
      editorPane.setEditable(false);

      HyperlinkEventInvoker firstLink =
         new HyperlinkEventInvoker(editorPane,
                                   HyperlinkEventInvoker.HYPERLINK_ENTERED,
                                   ui, "setBorder",
                                BorderFactory.createLineBorder(Color.red));
      firstLink.setUrlMatchPattern("link1");

      HyperlinkEventInvoker secondLink =
         new HyperlinkEventInvoker(editorPane,
                                   HyperlinkEventInvoker.HYPERLINK_ENTERED,
                                   ui, "setBorder",
                                     BorderFactory.createLineBorder(Color.green));
      secondLink.setUrlMatchPattern("link[23]");

      ui.add(editorPane);

      return ui;
   }

   private String getHTML() {
      return "<html><body>" +
                "<p>This contains <a href=\"link1\">a link</a></p>" +
                "<p>This contains <a href=\"link2\">another link</a></p>" +
                "<p>This contains <a href=\"link3\">yet another link</a></p>" +
                "</html></body>";
   }

   public static void main(String[] args) {
      new HyperlinkEventInvokerExample2();
   }
}
```

As you can see, the first `HyperlinkEventInvoker` triggers only on hover over the URL of "link1", but the second one triggers if the user hovers over either the link with the URL of "link2" or "link3". Most certainly, if the `HyperlinkEventInvoker` is targeting the activation actions, then the URL match pattern should match each link's URL independently.

InternalFrameEventInvoker

Internal frames produce quite a few events, and there may be many reasons to react to any of them. For example, if the frame is iconified, maybe an animation routine needs to pause until it is de-iconified. Handling of these events is the job `InternalFrameEventInvoker`. The masks for instances of this class are:

```
FRAME_ACTIVATED
FRAME_CLOSED
FRAME_CLOSING
FRAME_DEACTIVATED
FRAME_DEICONIFIED
FRAME_ICONIFIED
FRAME_OPENED
```

Again, these can be combined with the "|" character if you want to route the events to the same method:

```
import java.awt.Color;
import java.awt.Container;
import java.awt.Dimension;

import javax.swing.JDesktopPane;
import javax.swing.JInternalFrame;

import com.jmorgan.swing.JMFrame;
import com.jmorgan.swing.event.InternalFrameEventInvoker;

public class InternalFrameEventInvokerExample extends JMFrame {
   public InternalFrameEventInvokerExample() {
      super("InternalFrameEventInvokerExample");
   }

   public Container buildGUI() {
      JDesktopPane pane = new JDesktopPane();

      JInternalFrame iFrame = new JInternalFrame("Test Frame",
                                                  false, false, true, true);
      iFrame.setVisible(true);
      iFrame.setSize(200, 200);

      new InternalFrameEventInvoker(iFrame,
                                    InternalFrameEventInvoker.FRAME_ICONIFIED,
                                 pane, "setBackground", Color.RED);
      new InternalFrameEventInvoker(iFrame,
                                    InternalFrameEventInvoker.FRAME_DEICONIFIED,
                                 pane, "setBackground", Color.WHITE);

      pane.add(iFrame);
      pane.setPreferredSize(new Dimension(250, 250));

      return pane;
   }

   public static void main(String[] args) {
      new InternalFrameEventInvokerExample();
   }
}
```

The usage is quite obvious. Just figure out the best use case for your applications.

KeyEventInvoker

As its name sounds, `KeyEventInvoker` targets a method when one of the three events of the `KeyEvent` occurs. Like the `HyperLinkEventInvoker`, you can also filter based upon one or more key codes. By default, the target method is invoked based upon the designated masked event when any key is pressed. The masks for this class are:

```
KEY_TYPED
KEY_PRESSED
KEY_RELEASED
```

The below example shows both ways to setup the `KeyEventInvoker` with a key filter. Just keep in mind that if you just want to react to any key being pressed, don't define a key filter, or set the filter to an empty array:

```
import java.awt.BorderLayout;
import java.awt.Container;
import java.awt.Dimension;
import java.awt.event.KeyEvent;

import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;

import com.jmorgan.swing.JMFrame;
import com.jmorgan.swing.event.KeyEventInvoker;

public class KeyEventInvokerExample extends JMFrame {
   public KeyEventInvokerExample() {
      super("KeyEventInvokerExample");
   }

   public Container buildGUI() {
      JPanel ui = new JPanel(new BorderLayout());

      JTextField textField = new JTextField("Type \"a1b2c3\" here");
      JLabel label = new JLabel();

      KeyEventInvoker letters =
         new KeyEventInvoker(textField,
                             KeyEventInvoker.KEY_RELEASED,
                             label, "setText", "A letter was pressed");
      letters.setKeyFilter(new int[] { KeyEvent.VK_A,
                                       KeyEvent.VK_B,
                                       KeyEvent.VK_C });

      KeyEventInvoker numbers =
         new KeyEventInvoker(textField,
                             KeyEventInvoker.KEY_RELEASED,
                             label, "setText", "A number was pressed");
      numbers.addKeyFilter(KeyEvent.VK_1);
      numbers.addKeyFilter(KeyEvent.VK_2);
      numbers.addKeyFilter(KeyEvent.VK_3);

      ui.add(textField, BorderLayout.NORTH);
      ui.add(label, BorderLayout.CENTER);

      ui.setPreferredSize(new Dimension(200, 200));

      return ui;
   }

   public static void main(String[] args) {
      new KeyEventInvokerExample();
   }
}
```

MouseEventInvoker

The `MouseEventInvoker` is another that fires quite a few events. The mask options for the `MouseEventInvoker` are:

```
    MOUSE_CLICKED
    MOUSE_ENTERED
    MOUSE_EXITED
    MOUSE_PRESSED
    MOUSE_RELEASED
```

Like the other masked event invokers, the masks can be combined. Instances of this class also support an optional hot spot relative to the coordinates of the component. If no hot spot is defined, then anywhere the event occurs over the component is fair game. Any hot spot defined controls the limits of where the event is allowed to occur over the component. Note, too, that the hot spot is a `Shape`, which enables the most flexibility when defining a hot spot.

In this example, you will see an interesting combination of DEF elements and begin to experience a bit more of the power and flexibility of the DEF:

```
import java.awt.Container;
import java.awt.Dimension;
import java.awt.GridLayout;
import java.awt.Rectangle;

import javax.swing.JLabel;
import javax.swing.JPanel;

import com.jmorgan.lang.MethodInvoker;
import com.jmorgan.swing.JMFrame;
import com.jmorgan.swing.event.ComponentEventInvoker;
import com.jmorgan.swing.event.MouseEventInvoker;

public class MouseEventInvokerExample extends JMFrame {
   public MouseEventInvokerExample() {
      super("MouseEventInvokerExample");
   }

   public Container buildGUI() {
      JPanel ui = new JPanel(new GridLayout(1, 1));

      JLabel mouseTarget =
         new JLabel("Move the mouse and click on this label", JLabel.CENTER);

      new MouseEventInvoker(mouseTarget, MouseEventInvoker.MOUSE_ENTERED,
                            mouseTarget, "setText",
                            "Mouse Entered");
      new MouseEventInvoker(mouseTarget, MouseEventInvoker.MOUSE_EXITED,
                            mouseTarget, "setText",
                            "Mouse Exited");

      MouseEventInvoker left =
         new MouseEventInvoker(mouseTarget, MouseEventInvoker.MOUSE_CLICKED,
                               mouseTarget, "setText",
                               "Mouse Clicked In Left Side");
```

```
    MouseEventInvoker right =
      new MouseEventInvoker(mouseTarget, MouseEventInvoker.MOUSE_CLICKED,
                            mouseTarget, "setText",
                            "Mouse Clicked In Right Side");

    MethodInvoker mi = new MethodInvoker(mouseTarget, "getSize");
    new ComponentEventInvoker(mouseTarget,
                              ComponentEventInvoker.COMPONENT_RESIZED,
                              this, "setHotSpot", left, right, mi);

    ui.add(mouseTarget);

    ui.setPreferredSize(new Dimension(300, 100));

    return ui;
  }

  private void setHotSpot(MouseEventInvoker left,
                          MouseEventInvoker right,
                          Dimension shapeSize) {
    int halfWidth = shapeSize.width / 2;
    left.setHotSpot(new Rectangle(new Dimension(halfWidth, shapeSize.height)));
    right.setHotSpot(new Rectangle(halfWidth, 0, halfWidth, shapeSize.height));
  }

  public static void main(String[] args) {
    new MouseEventInvokerExample();
  }
}
```

Play a bit with this example. When you move the mouse into and out of the label, anywhere, you see the message “Mouse Entered” and “Mouse Exited”. However, only when you click on the left side of the label do you see the “Mouse Clicked In Left Side”, and only when you click on the right side of the label do you see the “Mouse Clicked In Right Side”. This is because there are two `MouseEventInvoker` instances, one with a hot spot defined to trigger when the mouse is only on the left, and one with a hot spot only on the right.

The hot spot shapes are defined when the label is resized, triggered by a `ComponentEventInvoker` that calls the `setHotSpot` method of the class. The `ComponentEventInvoker` passes references to the `MouseEventInvokers` for the left and right, and also passes the size of the component after it is resized. The `ComponentEventInvoker` obtains the current size of the component through the `MethodInvoker` referencing the `getSize` method of the label. This maintains the hot spots fully on the left and right regardless of the size of the component, rather than a fixed size at the time of construction.

MouseMotionEventInvoker

The `MouseMotionEventInvoker` is identical in concept to the `MouseEventInvoker`. This one only contains two mask values:

```
MOUSE_MOVED
MOUSE_DRAGGED
```

And just like the `MouseEventInvoker`, a `Shape` can be used to limit the area where the invoker will trigger. The example, again extremely similar to the example given for `MouseEventInvoker`, demonstrates the features of this class:

```
import java.awt.Container;
import java.awt.Dimension;
import java.awt.GridLayout;
import java.awt.Rectangle;

import javax.swing.JLabel;
import javax.swing.JPanel;

import com.jmorgan.lang.MethodInvoker;
import com.jmorgan.swing.JMFrame;
import com.jmorgan.swing.event.ComponentEventInvoker;
import com.jmorgan.swing.event.MouseMotionEventInvoker;

public class MouseMotionEventInvokerExample extends JMFrame {
   public MouseMotionEventInvokerExample() {
      super("MouseMotionEventInvokerExample");
   }

   public Container buildGUI() {
      JPanel ui = new JPanel(new GridLayout(1, 1));

      JLabel mouseTarget =
         new JLabel("Move the mouse and drag on this label", JLabel.CENTER);

      new MouseMotionEventInvoker(mouseTarget,
                                  MouseMotionEventInvoker.MOUSE_DRAGGED,
                                  mouseTarget, "setText",
                                  "Mouse Dragged");

      MouseMotionEventInvoker left =
         new MouseMotionEventInvoker(mouseTarget,
                                  MouseMotionEventInvoker.MOUSE_MOVED,
                                 mouseTarget, "setText",
                                 "Mouse Moved In Left Side");

      MouseMotionEventInvoker right =
         new MouseMotionEventInvoker(mouseTarget,
                                  MouseMotionEventInvoker.MOUSE_MOVED,
                                 mouseTarget, "setText",
                                 "Mouse Moved In Right Side");

      MethodInvoker mi = new MethodInvoker(mouseTarget, "getSize");
      new ComponentEventInvoker(mouseTarget,
                                   ComponentEventInvoker.COMPONENT_RESIZED,
                                   this, "setHotSpot", left, right, mi);
      ui.add(mouseTarget);

      ui.setPreferredSize(new Dimension(300, 100));

      return ui;
   }

   private void setHotSpot(MouseMotionEventInvoker left,
                           MouseMotionEventInvoker right,
                           Dimension shapeSize) {
      int halfWidth = shapeSize.width / 2;
      left.setHotSpot(new Rectangle(new Dimension(halfWidth, shapeSize.height)));
      right.setHotSpot(new Rectangle(halfWidth, 0, halfWidth, shapeSize.height));
   }

   public static void main(String[] args) {
      new MouseMotionEventInvokerExample();
   }
}
```

Extending AbstractEventInvoker and AbstractMaskedEventInvoker

In its current version, every event possible is not covered by the DEF. Being that you can roll your own events, you might want to be able to extend the DEF. A couple of things to note to help you conform to the development pattern these classes expect of an event producer.

Generally speaking, the pattern should mimic that of the delegation event mechanisms built into Java. There should be an event listener interface, an event object, and an event producer. The event producer should have a method allowing event listeners to be added to and removed from the producer. If you do that, the rest will fall into place nicely.

So, let's consider a scenario. Suppose you create a generic `Spinner` component that will allow any other component to be attached to it. The `Spinner` component will display an up arrow and a down arrow next to the component attached to it and produce events when the spin buttons are clicked. Since the spinner doesn't know exactly what it is spinning, it exposes a delegation event model having a `SpinListener` that has two methods, `spinUp` and `spinDown`.

Since the `Spinner` is the event producer, the class will contain two methods, `addSpinListener` and `removeSpinListener` to enable `SpinListeners` to be registered to it. Both methods take a reference to a `SpinListener` and perform the appropriate behavior relative to the semantics of the registration method.

When the up arrow is clicked, the `Spinner` creates a `SpinEvent` object and passes it to all registered `SpinListener`'s `spinUp` method. Likewise, when the down arrow is clicked, a `SpinEvent` is created and passed to all registered `SpinListener`'s `spinDown` method. So the `SpinListener` and `SpinEvent` classes might look something like this:

```
public interface SpinListener {
  /** Notification for a spin up.
           @param e A SpinEvent.
  */
  public void spinUp(SpinEvent e);

  /** Notification for a spin down.
           @param e A SpinEvent.
  */
  public void spinDown(SpinEvent e);
}

public class SpinEvent  {
  private Component source;
  private int spinDirection;

  /** Spin event constant indicating the the spin direction is up */
  public static final int SPINUP = 0;

  /** Spin event constant indicating the the spin direction is down */
  public static final int SPINDOWN = 1;

  /** Returns an instance to a SpinEvent
           @param source A source Component.
           @param spinDirection An int, 0 = SPINUP, 1 = SPINDOWN.
  */
  public SpinEvent(Component source, int spinDirection) {
    this.source = source;
    this.spinDirection = spinDirection;
  }

  /** Returns a number representing the spin direction.  The
      value matches either SpinEvent.SPINUP ( 0 ), or
      SpinEven.SPINDOWN ( 1 ).
           @return int The spin direction.
      @see #SPINUP
      @see #SPINDOWN
  */
  public int getSpinDirection() { return this.spinDirection; }

  /** Returns the source for which the spin is ocurring.
           @return Component The source component
  */
  public Component getSource() { return this.source; }
}
```

So, now you want to extend the DEF to support this new event. In the same pattern as exists within the DEF, maybe you decide to name the new invoker, `SpinEventInvoker`. Since the `SpinListener` interface contains more than one method, you should extend this class from `AbstractMaskedEventInvoker`, and define a couple of mask constants that can safely be OR'd together to combine a mask.

All event invokers need to know the names of the add and remove listener registration methods. In short, the `AbstractEventInvoker` provides a dynamic registration of its subclasses to the event producer, so your custom extensions will need to provide these names. The other thing the concrete extensions need to do is to implement the desired listener interface. So, your `SpinEventInvoker` will very likely look like this:

```
import com.jmorgan.swing.spinner.SpinEvent;
import com.jmorgan.swing.spinner.SpinListener;

public class SpinEventInvoker extends AbstractMaskedEventInvoker<SpinEvent>
                              implements SpinListener {
   public static final int SPIN_UP = 1;
   public static final int SPIN_DOWN = 2;

   public SpinEventInvoker(Object eventProducer, int eventMask,
                           Object target, String methodName) {
      super(eventProducer, eventMask, target, methodName);
      this.setListenerAddRemoveMethodNames("addSpinListener", "removeSpinListener");
   }

   public SpinEventInvoker(Object eventProducer, int eventMask,
                           Object target, String methodName, Object... arguments) {
      super(eventProducer, eventMask, target, methodName, arguments);
      this.setListenerAddRemoveMethodNames("addSpinListener", "removeSpinListener");
   }

   public void spinUp(SpinEvent e) {
      this.setEvent(e);
      this.invokeForEvent(SPIN_UP);
   }

   public void spinDown(SpinEvent e) {
      this.setEvent(e);
      this.invokeForEvent(SPIN_DOWN);
   }
}
```

And, that's it! Not really rocket science, and no wizardry. Very, very simple, but your new class now contains all of the basic features the event invokers within the DEF contains, including the ability to pass arguments to the target methods, and accept `MethodInvoker` instances as parameter value delegates for those arguments, all in just a handful of lines of code!

Chapter 8 – Things That Go Wrong

Oops!

There are many things that can go wrong, and thus there are some things you need know so you can tell what is happening within the guts of the DEF. The DEF makes every effort possible to locate the method you are after. When you give `AsynchMethodInvoker` or `MethodInvoker` the name of a method, it attempts to resolve the method reference during construction. However, when it cannot locate the method, it will throw a `RuntimeException` wrapped around a `NoSuchMethodException`. Here is a quick example:

```
import java.util.Calendar;

import com.jmorgan.lang.MethodInvoker;

public class NoSuchMethodExceptionExample {
   public static void main(String[] args) {
      Calendar c = Calendar.getInstance();
      new MethodInvoker(c, "BlahBlah");
   }
}
```

Running this throws this exception:

```
Exception in thread "main" java.lang.RuntimeException: The method BlahBlah does not
exist within java.util.GregorianCalendar
	at
com.jmorgan.lang.AbstractMethodInvoker.resolveMethod(AbstractMethodInvoker.java:183
)
	at
com.jmorgan.lang.AbstractMethodInvoker.<init>(AbstractMethodInvoker.java:128)
	at com.jmorgan.lang.MethodInvoker.<init>(MethodInvoker.java:99)
	at
defusermanualsamples.NoSuchMethodExceptionExample.main(NoSuchMethodExceptionExample
.java:10)
Caused by: java.lang.NoSuchMethodException: The method BlahBlah() does not exist
within java.lang.Object
	at
com.jmorgan.lang.AbstractMethodInvoker.getMethod(AbstractMethodInvoker.java:387)
	at
com.jmorgan.lang.AbstractMethodInvoker.getMethod(AbstractMethodInvoker.java:370)
	at
com.jmorgan.lang.AbstractMethodInvoker.getMethod(AbstractMethodInvoker.java:370)
	at
com.jmorgan.lang.AbstractMethodInvoker.resolveMethod(AbstractMethodInvoker.java:157
)
	... 3 more
```

The `NoSuchMethodException` gives the full message signature, but the incorrect class name for which the method is being resolved. The `RuntimeException` resolves the actual class name. This is because the actual method resolution algorithm will recursively search all ancestors all the way up through

Object. When that fails, it throws the NoSuchMethodException, and that always happens after failing to find the method in Object. This recursive ancestor search is the last-ditch effort to locate the method, so, upon receiving a NoSuchMethodException, reports the type of the initial target. This can be very helpful when you have a number of dynamic calls within a multi-threaded application, which happens quite often within an application heavily using the DEF, and definitely within UI applications.

You can help the DEF in some ways. First, AsynchMethodInvoker and MethodInvoker will accept either the name of the method as a String, or a direct reference to a Method object. Providing the pre-resolved Method reference bypasses the method resolution algorithm, thus preventing a NoSuchMethodException.

When providing the pre-resolved method reference, but you pass in the incorrect argument types for the method, you will get an IllegalArgumentException. Here is an example that attempts to pass a double instead of an int to a MethodInvoker with a pre-resolved Method:

```
import java.lang.reflect.Method;
import java.util.Calendar;

import com.jmorgan.lang.MethodInvoker;

public class IllegalArgumentExceptionExample {
   public static void main(String[] args) {
      Calendar c = Calendar.getInstance();
      try {
         Method targetMethod =
               c.getClass().getMethod("set",
                         new Class[] { int.class, int.class, int.class });
         new MethodInvoker(c, targetMethod, 0, 0.0, 0).invoke();
      }
      catch (Exception e) {
      }
   }
}
```

Obfuscation

Obfuscation tools can really mess this up, so you need to ensure all methods invoked through the DEF are kept. There is an annotation within the DEF, @Reflected, that will help you mark methods invoked by the DEF. Methods you cannot mark, such as those from a third-party API or those classes embodied within Java itself, that are the target of the DEF, should be kept as is if being changed by an obfuscator. If you are diligent, though, on your own methods, marking them with the @Reflected annotation will allow you to capture all of those in one place within your obfuscation tool.

Alphabetical Index