Regular Expressions and High-Performance I/O

Java™ NIO

O'REILLY®

Ron Hitchens

Java™ NIO

Ron Hitchens

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

CHAPTER 4

Selectors

Life is a series of rude awakenings.
—R. Van Winkle

In this chapter, we'll explore selectors. Selectors provide the ability to do *readiness selection*, which enables *multiplexed I/O*. As described in Chapter 1, readiness selection and multiplexing make it possible for a single thread to efficiently manage many I/O channels simultaneously. C/C++ coders have had the POSIX `select()` and/or `poll()` system calls in their toolbox for many years. Most other operating systems provide similar functionality. But readiness selection was never available to Java programmers until JDK 1.4. Programmers whose primary body of experience is in the Java environment may not have encountered this I/O model before.

For an illustration of readiness selection, let's return to the drive-through bank example of Chapter 3. Imagine a bank with three drive-through lanes. In the traditional (nonselector) scenario, imagine that each drive-through lane has a pneumatic tube that runs to its own teller station inside the bank, and each station is walled off from the others. This means that each tube (channel) requires a dedicated teller (worker thread). This approach doesn't scale well and is wasteful. For each new tube (channel) added, a new teller is required, along with associated overhead such as tables, chairs, paper clips (memory, CPU cycles, context switching), etc. And when things are slow, these resources (which have associated costs) tend to sit idle.

Now imagine a different scenario in which each pneumatic tube (channel) is connected to a single teller station inside the bank. The station has three slots where the carriers (data buffers) arrive, each with an indicator (selection key) that lights up when the carrier is in the slot. Also imagine that the teller (worker thread) has a sick cat and spends as much time as possible reading *Do It Yourself Taxidermy*.* At the end of each paragraph, the teller glances up at the indicator lights (invokes `select()`) to determine if any of the channels are ready (readiness selection). The teller (worker

* Not currently in the O'Reilly catalog.

thread) can perform another task while the drive-through lanes (channels) are idle yet still respond to them in a timely manner when they require attention.

While this analogy is not exact, it illustrates the paradigm of quickly checking to see if attention is required by any of a set of resources, without being forced to wait if something isn't ready to go. This ability to check and continue is key to scalability. A single thread can monitor large numbers of channels with readiness selection. The `Selector` and related classes provide the APIs to do readiness selection on channels.

Selector Basics

Getting a handle on the topics discussed in this chapter will be somewhat tougher than understanding the relatively straightforward buffer and channel classes. It's trickier, because there are three main classes, all of which come into play at the same time. If you find yourself confused, back up and take another run at it. Once you see how the pieces fit together and their individual roles, it should all make sense.

We'll begin with the executive summary, then break down the details. You register one or more previously created selectable channels with a selector object. A key that represents the relationship between one channel and one selector is returned. Selection keys remember what you are interested in for each channel. They also track the operations of interest that their channel is currently ready to perform. When you invoke `select()` on a selector object, the associated keys are updated by checking all the channels registered with that selector. You can obtain a set of the keys whose channels were found to be ready at that point. By iterating over these keys, you can service each channel that has become ready since the last time you invoked `select()`.

That's the 30,000-foot view. Now let's swoop in low and see what happens at ground level (or below).

At this point, you may want to skip ahead to Example 4-1 and take a quick look at the code. Between here and there, you'll learn the specifics of how these new classes work, but armed with just the high-level information in the preceding paragraph, you should be able to see how the selection model works in practice.

At the most fundamental level, selectors provide the capability to ask a channel if it's ready to perform an I/O operation of interest to you. For example, a `SocketChannel` object could be asked if it has any bytes ready to read, or we may want to know if a `ServerSocketChannel` has any incoming connections ready to accept.

`Selectors` provide this service when used in conjunction with `SelectableChannel` objects, but there's more to the story than that. The real power of readiness selection is that a potentially large number of channels can be checked for readiness simultaneously. The caller can easily determine which of several channels are ready to go. Optionally, the invoking thread can ask to be put to sleep until one or more of the channels registered with the `Selector` is ready, or it can periodically poll the

selector to see if anything has become ready since the last check. If you think of a web server, which must manage large numbers of concurrent connections, it's easy to imagine how these capabilities can be put to good use.

At first blush, it may seem possible to emulate readiness selection with nonblocking mode alone, but it really isn't. Nonblocking mode will either do what you request or indicate that it can't. This is semantically different from determining if it's *possible* to do a certain type of operation. For example, if you attempt a nonblocking read and it succeeds, you not only discovered that a `read()` is possible, you also read some data. You must then do something with that data.

This effectively prevents you from separating the code that checks for readiness from the code that processes the data, at least without significant complexity. And even if it was possible simply to ask each channel if it's ready, this would still be problematic because your code, or some code in a library package, would need to iterate through all the candidate channels and check each in turn. This would result in at least one system call per channel to test its readiness, which could be expensive, but the main problem is that the check would not be atomic. A channel early in the list could become ready after it's been checked, but you wouldn't know it until the next time you poll. Worst of all, you'd have no choice but to continually poll the list. You wouldn't have a way of being notified when a channel you're interested in becomes ready.

This is why the traditional Java solution to monitoring multiple sockets has been to create a thread for each and allow the thread to block in a `read()` until data is available. This effectively makes each blocked thread a socket monitor and the JVM's thread scheduler becomes the notification mechanism. Neither was designed for these purposes. The complexity and performance cost of managing all these threads, for the programmer and for the JVM, quickly get out of hand as the number of threads grows.

True readiness selection must be done by the operating system. One of the most important functions performed by an operating system is to handle I/O requests and notify processes when their data is ready. So it only makes sense to delegate this function down to the operating system. The `Selector` class provides the abstraction by which Java code can request readiness selection service from the underlying operating system in a portable way.

Let's take a look at the specific classes that deal with readiness selection in the `java.nio.channels` package.

The Selector, SelectableChannel, and SelectionKey Classes

At this point, you may be confused about how all this selection stuff works in Java. Let's identify the moving parts and how they interact. The UML diagram in Figure 4-1 makes the situation look more complicated than it really is. Refer to

Figure 4-2 and you'll see that there are really only three pertinent class APIs when doing readiness selection:

`Selector`
: The `Selector` class manages information about a set of registered channels and their readiness states. Channels are registered with selectors, and a selector can be asked to update the readiness states of the channels currently registered with it. When doing so, the invoking thread can optionally indicate that it would prefer to be suspended until one of the registered channels is ready.

`SelectableChannel`
: This abstract class provides the common methods needed to implement channel selectability. It's the superclass of all channel classes that support readiness selection. `FileChannel` objects are not selectable because they don't extend from `SelectableChannel` (see Figure 4-2). All the socket channel classes are selectable, as well as the channels obtained from a `Pipe` object. `SelectableChannel` objects can be registered with `Selector` objects, along with an indication of which operations on that channel are of interest for that selector. A channel can be registered with multiple selectors, but only once per selector.

`SelectionKey`
: A `SelectionKey` encapsulates the registration relationship between a specific channel and a specific selector. A `SelectionKey` object is returned from `SelectableChannel.register( )` and serves as a token representing the registration. `SelectionKey` objects contain two bit sets (encoded as integers) indicating which channel operations the registrant has an interest in and which operations the channel is ready to perform.

Let's take a look at the relevant API methods of `SelectableChannel`:

```
public abstract class SelectableChannel
        extends AbstractChannel
        implements Channel
{
        // This is a partial API listing

        public abstract SelectionKey register (Selector sel, int ops)
                throws ClosedChannelException;
        public abstract SelectionKey register (Selector sel, int ops, Object att)
                throws ClosedChannelException;

        public abstract boolean isRegistered( );
        public abstract SelectionKey keyFor (Selector sel);
        public abstract int validOps( );

        public abstract void configureBlocking (boolean block)
                throws IOException;
        public abstract boolean isBlocking( );
        public abstract Object blockingLock( );
}
```

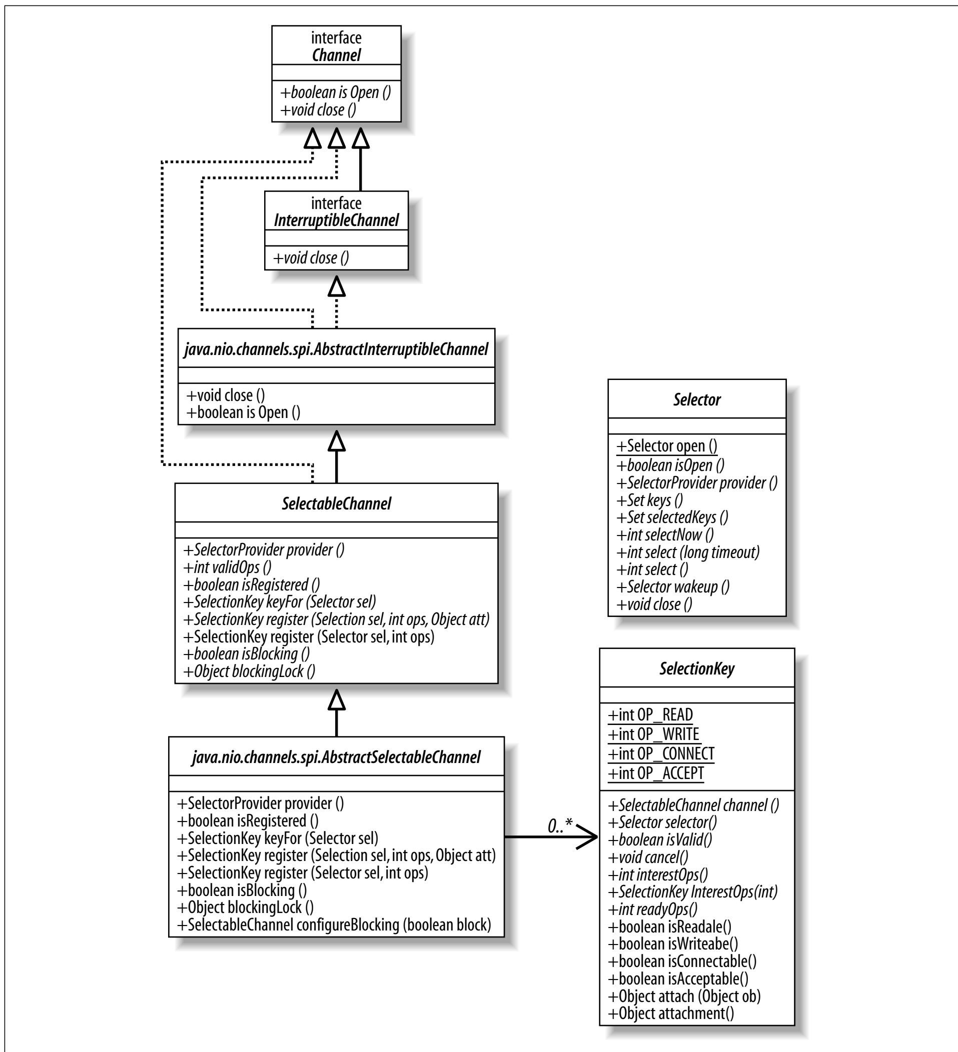


Figure 4-1. Selection class family tree

Nonblocking and multiplexing go hand-in-hand—so much so that the architects of `java.nio` placed the APIs for both in the same class.

We've already discussed how to configure and check a channel's blocking mode with the last three methods of `SelectableChannel`, which are listed above. (Refer to "Nonblocking Mode" in Chapter 3 for a detailed discussion.) A channel must first be placed in nonblocking mode (by calling `configureBlocking(false)`) before it can be registered with a selector.

Invoking the selectable channel's `register()` method registers it with a selector. If you attempt to register a channel that is in blocking mode, `register()` will throw an

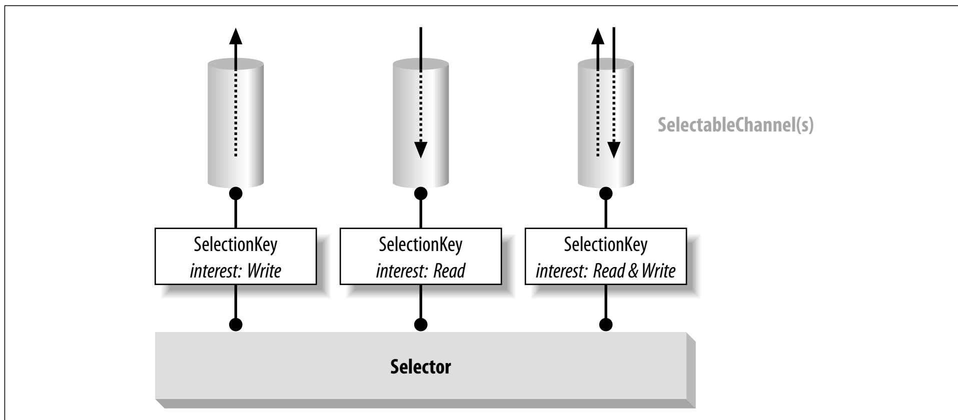


Figure 4-2. Relationships of the selection classes

unchecked `IllegalBlockingModeException`. Also, a channel cannot be returned to blocking mode while registered. Attempting to do so will throw `IllegalBlockingModeException` from the `configureBlocking( )` method.

And, of course, attempting to register a `SelectableChannel` instance that has been closed will throw `ClosedChannelException`, as indicated by the method signature.

Before we take a closer look at `register()` and the other methods of `SelectableChannel`, let's look at the API of the `Selector` class so we can better understand the relationship:

```
public abstract class Selector
{
        public static Selector open( ) throws IOException
        public abstract boolean isOpen( );
        public abstract void close( ) throws IOException;
        public abstract SelectionProvider provider( );

        public abstract int select( ) throws IOException;
        public abstract int select (long timeout) throws IOException;
        public abstract int selectNow( ) throws IOException;
        public abstract void wakeup( );

        public abstract Set keys( );
        public abstract Set selectedKeys( );
}
```

Although the `register( )` method is defined on the `SelectableChannel` class, channels are registered with selectors, not the other way around. A selector maintains a set of channels to monitor. A given channel can be registered with more than one selector and has no idea which `Selector` objects it's currently registered with. The choice to put the `register( )` method in `SelectableChannel` rather than in `Selector` was somewhat arbitrary. It returns a `SelectionKey` object that encapsulates a relationship

between the two objects. The important thing is to remember that the Selector object controls the selection process for the channels registered with it.

```
    public abstract class SelectionKey
    {
        public static final int OP_READ
        public static final int OP_WRITE
        public static final int OP_CONNECT
        public static final int OP_ACCEPT

        public abstract SelectableChannel channel( );
        public abstract Selector selector( );

        public abstract void cancel( );
        public abstract boolean isValid( );

        public abstract int interestOps( );
        public abstract void interestOps (int ops);
        public abstract int readyOps( );

        public final boolean isReadable( )
        public final boolean isWritable( )
        public final boolean isConnectable( )
        public final boolean isAcceptable( )

        public final Object attach (Object ob)
        public final Object attachment( )
}
```

Selectors are the managing objects, not the selectable channel objects. The Selector object performs readiness selection of channels registered with it and manages selection keys.

The interpretation of a key's interest and readiness sets is channel-specific. Each channel implementation typically defines its own specialized SelectionKey class, constructs it within the register( ) method, and passes it to the provided Selector object.

In the following sections, we'll cover all the methods of these three classes in more detail.

Setting Up Selectors

At this point, you may still be confused. You see a bunch of methods in the three preceding listings and can't tell what they do or what they mean. Before drilling into the details of all this, let's take a look at a typical usage example. It should help to put things into context.

To set up a Selector to monitor three Socket channels, you'd do something like this (refer to Figure 4-2):

```
Selector selector = Selector.open( );

channel1.register (selector, SelectionKey.OP_READ);
channel2.register (selector, SelectionKey.OP_WRITE);
channel3.register (selector, SelectionKey.OP_READ | SelectionKey.OP_WRITE);

// Wait up to 10 seconds for a channel to become ready
readyCount = selector.select (10000);
```

This code creates a new selector, then registers three (preexisting) socket channels with that selector, each with a different set of interests. The `select( )` method is then called to put the thread to sleep until one of these interesting things happens or the 10-second timer expires.

Now let's start looking at the `Selector` API in detail:

```
public abstract class Selector
{
        // This is a partial API listing

        public static Selector open( ) throws IOException
        public abstract boolean isOpen( );
        public abstract void close( ) throws IOException;
        public abstract SelectionProvider provider( );
}
```

Selector objects are instantiated by calling the static factory method `open( )`. Selectors are not primary I/O objects like channels or streams: data never passes through them. The `open( )` class method interfaces to the SPI to request a new instance from the default `SelectorProvider` object. It's also possible to create a new `Selector` instance by calling the `openSelector( )` method of a custom `SelectorProvider` object. You can determine which `SelectorProvider` object created a given `Selector` instance by calling its `provider( )` method. In most cases, you do not need to be concerned about the SPI; just call `open( )` to create a new `Selector` object. For those rare circumstances when you must deal with it, the channel SPI package is summarized in Appendix B.

Continuing the convention of treating a `Selector` as an I/O object: when you're finished with it, call `close( )` to release any resources it may be holding and to invalidate any associated selection keys. Once a `Selector` has been closed, attempting to invoke most methods on it will result in a `ClosedSelectorException`. Note that `ClosedSelectorException` is an unchecked (runtime) exception. You can test a `Selector` to determine if it's currently open with the `isOpen( )` method.

We'll finish with the `Selector` API in a bit, but right now let's take a look at registering channels with selectors. Here's an abbreviated version of the `SelectableChannel` API from earlier in this chapter:

```
public abstract class SelectableChannel
        extends AbstractChannel
        implements Channel
```

```
{
        // This is a partial API listing

        public abstract SelectionKey register (Selector sel, int ops)
                throws ClosedChannelException;
        public abstract SelectionKey register (Selector sel, int ops, Object att)
                throws ClosedChannelException;

        public abstract boolean isRegistered( );
        public abstract SelectionKey keyFor (Selector sel);
        public abstract int validOps( );
}
```

As mentioned earlier, the `register( )` method lives in the `SelectableChannel` class, although channels are actually registered with selectors. You can see that `register( )` takes a `Selector` object as an argument, as well as an integer parameter named `ops`. This second argument represents the *operation interest set* for which the channel is being registered. This is a bit mask that represents the I/O operations that the selector should test for when checking the readiness of that channel. The specific operation bits are defined as public static fields in the `SelectionKey` class.

As of JDK 1.4, there are four defined selectable operations: read, write, connect, and accept. Not all operations are supported on all selectable channels. A `SocketChannel` cannot do an accept, for example. Attempting to register interest in an unsupported operation will result in the unchecked `IllegalArgumentException` being thrown. You can discover the set of operations a particular channel object supports by calling its `validOps( )` method. We saw this method on the socket channel classes discussed in Chapter 3.

Selectors contain sets of channels currently registered with them. Only one registration of a given channel with a given selector can be in effect at any given time. However, it is permissible to register a given channel with a given selector more than once. Doing so returns the same `SelectionKey` object after updating its operation interest set to the given value. In effect, subsequent registrations simply update the key associated with the preexisting registration (see the section "Using Selection Keys").

An exceptional situation is when you attempt to reregister a channel with a selector for which the associated key has been cancelled, but the channel is still registered. Channels are not immediately deregistered when the associated key is cancelled. They remain registered until the next selection operation occurs (see the section "Using Selectors"). In this case, the unchecked `CancelledKeyException` will be thrown. Test the state of the `SelectionKey` object if there is a chance the key may have been cancelled.

In the previous listing, you'll notice a second version of `register( )` that takes a generic object argument. This is a convenience method that passes the object reference you

provide to the `attach()` method of the new selection key before returning it to you. We'll take a closer look at the API for `SelectionKey` in the next section.

A single channel object can be registered with multiple selectors. A channel can be queried to see if it is currently registered with any selectors by calling the `isRegistered()` method. This method does not provide information about which selectors the channel is registered with, only that it is registered with at least one. Additionally, there can be a delay between the time a registration key is cancelled and the time a channel is deregistered. This method is a hint, not a definitive answer.

Each registration of a channel with a selector is encapsulated by a `SelectionKey` object. The `keyFor()` method returns the key associated with this channel and the given selector. If the channel is currently registered with the given selector, the associated key is returned. If no current registration relationship exists for this channel with the given selector, `null` is returned.

Using Selection Keys

Let's look again at the API of the `SelectionKey` class:

```
package java.nio.channels;

public abstract class SelectionKey
{
        public static final int OP_READ
        public static final int OP_WRITE
        public static final int OP_CONNECT
        public static final int OP_ACCEPT

        public abstract SelectableChannel channel( );
        public abstract Selector selector( );

        public abstract void cancel( );
        public abstract boolean isValid( );

        public abstract int interestOps( );
        public abstract void interestOps (int ops);
        public abstract int readyOps( );

        public final boolean isReadable( )
        public final boolean isWritable( )
        public final boolean isConnectable( )
        public final boolean isAcceptable( )

        public final Object attach (Object ob)
        public final Object attachment( )
}
```

As mentioned earlier, a key represents the registration of a particular channel object with a particular selector object. You can see that relationship reflected in the first

two methods above. The channel( ) method returns the SelectableChannel object associated with the key, and selector( ) returns the associated Selector object. Nothing surprising there.

Key objects represent a specific registration relationship. When it's time to terminate that relationship, call the cancel( ) method on the SelectionKey object. A key can be checked to see if it still represents a valid registration by calling its isValid( ) method. When a key is cancelled, it's placed in the cancelled set of the associated selector. The registration is not immediately terminated, but the key is immediately invalidated (see the section "Using Selectors"). Upon the next invocation of select( ) (or upon completion of an in-progress select( ) invocation), any cancelled keys will be cleared from the cancelled key set, and the corresponding deregistrations will be completed. The channel can be reregistered, and a new SelectionKey object will be returned.

When a channel is closed, all keys associated with it are automatically cancelled (remember, a channel can be registered with many selectors). When a selector is closed, all channels registered with that selector are deregistered, and the associated keys are invalidated (cancelled). Once a key has been invalidated, calling any of its methods related to selection will throw a CancelledKeyException.

A SelectionKey object contains two sets encoded as integer bit masks: one for those operations of interest to the channel/selector combination (the *interest* set) and one representing operations the channel is currently ready to perform (the *ready* set). The current interest set can be retrieved from the key object by invoking its interestOps( ) method. Initially, this will be the value passed in when the channel was registered. This interest set will never be changed by the selector, but you can change it by calling interestOps( ) with a new bit mask argument. The interest set can also be modified by reregistering the channel with the selector (which is effectively a roundabout way of invoking interestOps( )), as described in the section "Setting Up Selectors." Changes made to the interest set of a key while a select( ) is in progress on the associated Selector will not affect that selection operation. Any changes will be seen on the next invocation of select( ).

The set of operations that are ready on the channel associated with a key can be retrieved by calling the key's readyOps( ) method. The ready set is a subset of the interest set and represents those operations from the interest set which were determined to be ready on the channel by the last invocation of select( ). For example, the following code tests to see if the channel associated with a key is ready for reading. If so, it reads data from it into a buffer and sends it along to a consumer method.

```
if ((key.readyOps( ) & SelectionKey.OP_READ) != 0)
{
        myBuffer.clear( );
        key.channel( ).read (myBuffer);
        doSomethingWithBuffer (myBuffer.flip( ));
}
```

As noted earlier, there are currently four channel operations that can be tested for readiness. You can check these by testing the bit mask as shown in the code above, but the `SelectionKey` class defines four boolean convenience methods to test the bits for you: `isReadable( )`, `isWritable( )`, `isConnectable( )`, and `isAcceptable( )`. Each of these is equivalent to checking the result of `readyOps( )` against the appropriate operation bit value. For example:

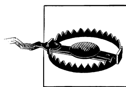
```
if (key.isWritable( ))
```

is equivalent to:

```
if ((key.readyOps( ) & SelectionKey.OP_WRITE) != 0)
```

All four of these methods are safe to call on any `SelectionKey` object. Recall that a channel cannot be registered for interest in an operation it doesn't support. Since an unsupported operation will never be in a channel's interest set, it can never appear in its ready set. Therefore, calling one of these methods for an unsupported operation will always return `false` because that operation will never be ready on that channel.

It's important to note that the readiness indication associated with a selection key as returned by `readyOps( )` is a hint, not an iron-clad guarantee. The state of the underlying channel can change at any time. Other threads may perform operations on the channel that affect its readiness state. And, as always, operating system–specific idiosyncrasies may come into play.



The ready set contained by a `SelectionKey` object is as of the time the selector last checked the states of the registered channels. The readiness of individual channels could have changed in the meantime.

You may have noticed from the `SelectionKey` API that although there is a way to get the operation ready set, there is no API method to set or reset the members of that set. You cannot, in fact, directly modify a key's ready set. In the next section, which describes the selection process, we'll see how selectors and keys interact to provide up-to-date readiness indication.

Let's examine the remaining two methods of the `SelectionKey` API:

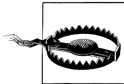
```
public abstract class SelectionKey
{
        // This is a partial API listing

        public final Object attach (Object ob)
        public final Object attachment( )
}
```

These two methods allow you to place an *attachment* on a key and retrieve it later. This is a convenience that allows you to associate an arbitrary object with a key. This object can be a reference to anything meaningful to you, such as a business object,

session handle, another channel, etc. This allows you to iterate through the keys associated with a selector, using the attached object handle on each as a reference to retrieve the associated context.

The `attach()` method stores the provided object reference in the key object. The `SelectionKey` class does not use the object except to store it. Any previous attachment reference stored in the key will be replaced. The `null` value may be given to clear the attachment. The attachment handle associated with a key can be fetched by calling the `attachment()` method. This method could return `null` if no attachment was set or if `null` was explicitly given.



If the selection key is long-lived, but the object you attach should not be, remember to clear the attachment when you're done. Otherwise, your attached object will not be garbage collected, and you may have a memory leak.

An overloaded version of the `register()` method on the `SelectableChannel` class takes an `Object` argument. This is a convenience that lets you attach an object to the new key during registration. This:

```
SelectionKey key = channel.register (selector, SelectionKey.OP_READ, myObject);
```

is equivalent to this:

```
SelectionKey key = channel.register (selector, SelectionKey.OP_READ);
key.attach (myObject);
```

One last thing to note about the `SelectionKey` class relates to concurrency. Generally, `SelectionKey` objects are thread-safe, but it's important to know that operations that modify the interest set are synchronized by `Selector` objects. This could cause calls to the `interestOps()` method to block for an indeterminate amount of time. The specific locking policy used by a selector, such as whether the locks are held throughout the selection process, is implementation-dependent. Luckily, this multiplexing capability is specifically designed to enable a single thread to manage many channels. Using selectors by multiple threads should be an issue in only the most complex of applications. Frankly, if you're sharing selectors among many threads and encountering synchronization issues, your design probably needs a rethink.

We've covered the API for the `SelectionKey` class, but we're not finished with selection keys—not by a long shot. Let's take a look at how to manage keys when using them with selectors.

Using Selectors

Now that we have a pretty good handle on the various classes and how they relate to one another, let's take a closer look at the `Selector` class, the heart of readiness

selection. Here is the abbreviated API of the `Selector` class we saw earlier. In the section "Setting Up Selectors," we saw how to create new selectors, so those methods have been left out:

```
public abstract class Selector
{
        // This is a partial API listing

        public abstract Set keys( );
        public abstract Set selectedKeys( );

        public abstract int select( ) throws IOException;
        public abstract int select (long timeout) throws IOException;
        public abstract int selectNow( ) throws IOException;

        public abstract void wakeup( );
}
```

The Selection Process

Before getting into the details of the API, you should know a little about the inner workings of `Selector`. As previously discussed, a selector maintains a set of registered channels, and each of these registrations is encapsulated in a `SelectionKey` object. Each `Selector` object maintains three sets of keys:

Registered key set
: The set of currently registered keys associated with the selector. Not every registered key is necessarily still valid. This set is returned by the `keys( )` method and may be empty. The registered key set is not directly modifiable; attempting to do so yields a `java.lang.UnsupportedOperationException`.

Selected key set
: A subset of the registered key set. Each member of this set is a key whose associated channel was determined by the selector (during a prior selection operation) to be ready for at least one of the operations in the key's interest set. This set is returned by the `selectedKeys( )` method (and may be empty).

  Don't confuse the selected key set with the ready set. This is a set of keys, each with an associated channel that is ready for at least one operation. Each key has an embedded ready set that indicates the set of operations the associated channel is ready to perform.

  Keys can be directly removed from this set, but not added. Attempting to add to the selected key set throws `java.lang.UnsupportedOperationException`.

Cancelled key set
: A subset of the registered key set, this set contains keys whose `cancel( )` methods have been called (the key has been invalidated), but they have not been deregistered. This set is private to the selector object and cannot be accessed directly.

All three of these sets are empty in a newly instantiated `Selector` object.

The core of the `Selector` class is the *selection* process. You've seen several references to it already—now it's time to explain it. Essentially, selectors are a wrapper for a native call to `select()`, `poll()`, or a similar operating system–specific system call. But the `Selector` does more than a simple pass-through to native code. It applies a specific process on each selection operation. An understanding of this process is essential to properly managing keys and the state information they represent.

A selection operation is performed by a selector when one of the three forms of `select()` is invoked. Whichever is called, the following three steps are performed:

1. The cancelled key set is checked. If it's nonempty, each key in the cancelled set is removed from the other two sets, and the channel associated with the cancelled key is deregistered. When this step is complete, the cancelled key set is empty.
2. The operation interest sets of each key in the registered key set are examined. Changes made to the interest sets after they've been examined in this step will not be seen during the remainder of the selection operation.

   Once readiness criteria have been determined, the underlying operating system is queried to determine the actual readiness state of each channel for its operations of interest. Depending on the specific `select()` method called, the thread may block at this point if no channels are currently ready, possibly with a timeout value.

   Upon completion of the system calls, which may have caused the invoking thread to be put to sleep for a while, the current readiness status of each channel will have been determined. Nothing further happens to any channel not found to be currently ready. For each channel that the operating system indicates is ready for at least one of the operations in its interest set, one of the following two things happens:

   a. If the key for the channel is not already in the selected key set, the key's ready set is cleared, and the bits representing the operations determined to be currently ready on the channel are set.
   b. Otherwise, the key is already in the selected key set. The key's ready set is updated by setting bits representing the operations found to be currently ready. Any previously set bits representing operations that are no longer ready are not cleared. In fact, no bits are cleared. The ready set as determined by the operating system is bitwise-disjoined into the previous ready set.[*] Once a key has been placed in the selected key set of the selector, its ready set is cumulative. Bits are set but never cleared.
3. Step 2 can potentially take a long time, especially if the invoking thread sleeps. Keys associated with this selector could have been cancelled in the meantime.

[*] A fancy way of saying the bits are logically ORed together.

When Step 2 completes, the actions taken in Step 1 are repeated to complete deregistration of any channels whose keys were cancelled while the selection operation was in progress.

4. The value returned by the select operation is the number of keys whose operation ready sets were modified in Step 2, not the total number of channels in the selection key set. The return value is not a count of ready channels, but the number of channels that became ready since the last invocation of `select()`. A channel ready on a previous call and still ready on this call won't be counted, nor will a channel that was ready on a previous call but is no longer ready. These channels could still be in the selection key set but *will not* be counted in the return value. The return value could be `0`.

Using the internal cancelled key set to defer deregistration is an optimization to prevent threads from blocking when they cancel a key and to prevent collisions with in-progress selection operations. Deregistering a channel is a potentially expensive operation that may require deallocation of resources (remember that keys are channel-specific and may have complex interactions with their associated channel objects). Cleaning up cancelled keys and deregistering channels immediately before or after a selection operation eliminates the potentially thorny problem of deregistering channels while they're in the middle of selection. This is another good example of compromise in favor of robustness.

The `Selector` class's `select()` method comes in three different forms:

```
public abstract class Selector
{
        // This is a partial API listing

        public abstract int select( ) throws IOException;
        public abstract int select (long timeout) throws IOException;
        public abstract int selectNow( ) throws IOException;

        public abstract void wakeup( );
}
```

The three forms of select differ only in whether they block if none of the registered channels are currently ready. The simplest form takes no argument and is invoked like this:

```
int n = selector.select( );
```

This call blocks indefinitely if no channels are ready. As soon as at least one of the registered channels is ready, the selection key set of the selector is updated, and the ready sets for each ready channel will be updated. The return value will be the number of channels determined to be ready. Normally, this method returns a nonzero value since it blocks until a channel is ready. But it can return `0` if the `wakeup()` method of the selector is invoked by another thread.

Sometimes you want to limit the amount of time a thread will wait for a channel to become ready. For those situations, an overloaded form of `select( )` that takes a timeout argument is available:

```
int n = selector.select (10000);
```

This call behaves exactly the same as the previous example, except that it returns a value of `0` if no channels have become ready within the timeout period you provide (specified in milliseconds). If one or more channels become ready before the time limit expires, the status of the keys will be updated, and the method will return at that point. Specifying a timeout value of `0` indicates to wait indefinitely and is identical in all respects to the no-argument version of `select( )`.

The third and final form of selection is totally nonblocking:

```
int n = selector.selectNow( );
```

The `selectNow( )` method performs the readiness selection process but will never block. If no channels are currently ready, it immediately returns `0`.

Stopping the Selection Process

The last of the `Selector` API methods, `wakeup( )`, provides the capability to gracefully break out a thread from a blocked `select( )` invocation:

```
public abstract class Selector
{
        // This is a partial API listing

        public abstract void wakeup( );
}
```

There are three ways to wake up a thread sleeping in `select( )`:

Call `wakeup( )`

Calling `wakeup( )` on a `Selector` object causes the first selection operation on that selector that has not yet returned to return immediately. If no selection is currently underway, then the next invocation of one of the `select( )` methods will return immediately. Subsequent selection operations will behave normally. Invoking `wakeup( )` multiple times between selection operations is no different than invoking it once.

Sometimes this deferred wakeup behavior may not be what you want. You may want to wake only a sleeping thread but allow subsequent selections to proceed normally. You can work around this problem by invoking `selectNow( )` after calling `wakeup( )`. However, if you structure your code to pay attention to the return codes and process the selection set properly, it shouldn't make any difference if the next `select( )` returns immediately with nothing ready. You should be prepared for this eventuality anyway.

Call `close()`
: If a selector's `close()` method is called, any thread blocked in a selection operation will be awakened as if the `wakeup()` method had been called. Channels associated with the selector will then be deregistered and the keys cancelled.

Call `interrupt()`
: If the sleeping thread's `interrupt()` method is called, its *interrupt status* is set. If the awakened thread then attempts an I/O operation on a channel, the channel is closed immediately, and the thread catches an exception. This is because of the interruption semantics of channels discussed in Chapter 3. Use `wakeup()` to gracefully awaken a thread sleeping in `select()`. Take steps to clear the interrupt status if you want a sleeping thread to continue after being directly interrupted (see the documentation for `Thread.interrupted()`).

  The `Selector` object catches the `InterruptedException` exception and call `wakeup()`.

Note that none of these methods automatically close any of the channels involved. Interrupting a selector is not the same as interrupting a channel (see "Closing Channels" in Chapter 3). Selection does not change the state of any of the channels involved, it only tests their state. There is no ambiguity regarding channel state when a thread sleeping in a selector is interrupted.

Managing Selection Keys

Now that we understand how the various pieces of the puzzle fit together, it's time see how they interoperate in normal use. To use the information provided by selectors and keys effectively, it's important to properly manage the keys.

Selections are cumulative. Once a selector adds a key to the selected key set, it never removes it. And once a key is in the selected key set, ready indications in the ready set of that key are set but never cleared. At first blush, this seems troublesome because a selection operation may not give a true representation of the current state of the registered channels. This is an intentional design decision. It provides a great deal of flexibility but assigns responsibility to the programmer to properly manage the keys to ensure that the state information they represent does not become stale.

The secret to using selectors properly is to understand the role of the selected key set maintained by the selector. (See the section "The Selection Process," specifically Step 2 of the selection process.) The important part is what happens when a key is *not* already in the selected set. When at least one operation of interest becomes ready on the channel, the ready set of the key is cleared, and the currently ready operations are added to the ready set. The key is then added to the selected key set.

The way to clear the ready set of a `SelectionKey` is to remove the key itself from the set of selected keys. The ready set of a selection key is modified only by the `Selector` object during a selection operation. The idea is that only keys in the selected set are

considered to have legitimate readiness information. That information persists in the key until the key is removed from the selected key set, which indicates to the selector that you have seen and dealt with it. The next time something of interest happens on the channel, the key will be set to reflect the state of the channel at that point and once again be added to the selected key set.

This scheme provides a lot of flexibility. The conventional approach is to perform a `select( )` call on the selector (which updates the selected key set) then iterate over the set of keys returned by `selectedKeys( )`. As each key is examined in turn, the associated channel is dealt with according to the key's ready set. The key is then removed from the selected key set (by calling `remove( )` on the `Iterator` object), and the next key is examined. When complete, the cycle repeats by calling `select( )` again. The code in Example 4-1 is a typical server example.

Example 4-1. Using select() to service multiple channels

```
package com.ronsoft.books.nio.channels;

import java.nio.ByteBuffer;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.nio.channels.Selector;
import java.nio.channels.SelectionKey;
import java.nio.channels.SelectableChannel;

import java.net.Socket;
import java.net.ServerSocket;
import java.net.InetSocketAddress;
import java.util.Iterator;

/**
 * Simple echo-back server which listens for incoming stream connections
 * and echoes back whatever it reads.  A single Selector object is used to
 * listen to the server socket (to accept new connections) and all the
 * active socket channels.
 *
 * @author Ron Hitchens (ron@ronsoft.com)
 */
public class SelectSockets
{
    public static int PORT_NUMBER = 1234;

    public static void main (String [] argv)
        throws Exception
    {
        new SelectSockets( ).go (argv);
    }

    public void go (String [] argv)
        throws Exception
    {
        int port = PORT_NUMBER;
```

Example 4-1. Using select() to service multiple channels (continued)

```
        if (argv.length > 0) {    // Override default listen port
            port = Integer.parseInt (argv [0]);
        }

        System.out.println ("Listening on port " + port);

        // Allocate an unbound server socket channel
        ServerSocketChannel serverChannel = ServerSocketChannel.open( );
        // Get the associated ServerSocket to bind it with
        ServerSocket serverSocket = serverChannel.socket( );
        // Create a new Selector for use below
        Selector selector = Selector.open( );

        // Set the port the server channel will listen to
        serverSocket.bind (new InetSocketAddress (port));

        // Set nonblocking mode for the listening socket
        serverChannel.configureBlocking (false);

        // Register the ServerSocketChannel with the Selector
        serverChannel.register (selector, SelectionKey.OP_ACCEPT);

        while (true) {
            // This may block for a long time. Upon returning, the
            // selected set contains keys of the ready channels.
            int n = selector.select( );

            if (n == 0) {
                continue;    // nothing to do
            }

            // Get an iterator over the set of selected keys
            Iterator it = selector.selectedKeys().iterator( );

            // Look at each key in the selected set
            while (it.hasNext( )) {
                SelectionKey key = (SelectionKey) it.next( );

                // Is a new connection coming in?
                if (key.isAcceptable( )) {
                    ServerSocketChannel server =
                        (ServerSocketChannel) key.channel( );
                    SocketChannel channel = server.accept( );

                    registerChannel (selector, channel,
                        SelectionKey.OP_READ);

                    sayHello (channel);
                }

                // Is there data to read on this channel?
                if (key.isReadable( )) {
```

Example 4-1. Using select() to service multiple channels (continued)

```
                    readDataFromSocket (key);
                }

                // Remove key from selected set; it's been handled
                it.remove( );
            }
        }
    }

    // ----------------------------------------------------------

    /**
     * Register the given channel with the given selector for
     * the given operations of interest
     */
    protected void registerChannel (Selector selector,
        SelectableChannel channel, int ops)
        throws Exception
    {
        if (channel == null) {
            return;        // could happen
        }

        // Set the new channel nonblocking
        channel.configureBlocking (false);

        // Register it with the selector
        channel.register (selector, ops);
    }

    // ----------------------------------------------------------

    // Use the same byte buffer for all channels.  A single thread is
    // servicing all the channels, so no danger of concurrent acccess.
    private ByteBuffer buffer = ByteBuffer.allocateDirect (1024);

    /**
     * Sample data handler method for a channel with data ready to read.
     * @param key A SelectionKey object associated with a channel
     *  determined by the selector to be ready for reading.  If the
     *  channel returns an EOF condition, it is closed here, which
     *  automatically invalidates the associated key.  The selector
     *  will then de-register the channel on the next select call.
     */
    protected void readDataFromSocket (SelectionKey key)
        throws Exception
    {
        SocketChannel socketChannel = (SocketChannel) key.channel( );
        int count;

        buffer.clear( );            // Empty buffer
```

Example 4-1. Using select() to service multiple channels (continued)

```
        // Loop while data is available; channel is nonblocking
        while ((count = socketChannel.read (buffer)) > 0) {
            buffer.flip( );        // Make buffer readable

            // Send the data; don't assume it goes all at once
            while (buffer.hasRemaining( )) {
                socketChannel.write (buffer);
            }
            // WARNING: the above loop is evil.  Because
            // it's writing back to the same nonblocking
            // channel it read the data from, this code can
            // potentially spin in a busy loop.  In real life
            // you'd do something more useful than this.

            buffer.clear( );        // Empty buffer
        }

        if (count < 0) {
            // Close channel on EOF, invalidates the key
            socketChannel.close( );
        }
    }

    // ----------------------------------------------------------

    /**
     * Spew a greeting to the incoming client connection.
     * @param channel The newly connected SocketChannel to say hello to.
     */
    private void sayHello (SocketChannel channel)
        throws Exception
    {
        buffer.clear( );
        buffer.put ("Hi there!\r\n".getBytes( ));
        buffer.flip( );

        channel.write (buffer);
    }

}
```

Example 4-1 implements a simple server. It creates `ServerSocketChannel` and `Selector` objects and registers the channel with the selector. We don't bother saving a reference to the registration key for the server socket because it will never be deregistered. The infinite loop calls `select( )` at the top, which may block indefinitely. When selection is complete, the selected key set is iterated to check for ready channels.

If a key indicates that its channel is ready to do an `accept( )`, we obtain the channel associated with the key and cast it to a `ServerSocketChannel` object. We know it's safe to do this because only `ServerSocketChannel` objects support the `OP_ACCEPT` operation. We also know our code registers only a single `ServerSocketChannel` object with

interest in OP_ACCEPT. With a reference to the server socket channel, we invoke accept( ) on it to obtain a handle to the incoming socket. The object returned is of type SocketChannel, which is also a selectable type of channel. At this point, rather than spawning a new thread to read data from the new connection, we simply register the socket channel with the selector. We tell the selector we're interested in knowing when the new socket channel is ready for reading by passing in the OP_READ flag.

If the key did not indicate that the channel was ready for accept, we check to see if it's ready for read. Any socket channels indicating so will be one of the SocketChannel objects previously created by the ServerSocketChannel and registered for interest in reading. For each socket channel with data to read, we invoke a common routine to read and process the data socket. Note that this routine should be prepared to deal with incomplete data on the socket, which is in nonblocking mode. It should return promptly so that other channels with pending input can be serviced in a timely manner. Example 4-1 simply echoes the data back down the socket to the sender.

At the bottom of the loop, we remove the key from the selected key set by calling remove( ) on the Iterator object. Keys can be removed directly from the Set returned by selectedKeys( ), but when examining the set with an Iterator, you should use the iterator's remove( ) method to avoid corrupting the iterator's internal state.

Concurrency

Selector objects are thread-safe, but the key sets they contain are not. The key sets returned by the keys( ) and selectedKeys( ) methods are direct references to private Set objects inside the Selector object. These sets can change at any time. The registered key set is read-only. If you attempt to modify it, your reward will be a java.lang.UnsupportedOperationException, but you can still run into trouble if it's changed while you're looking at it. Iterator objects are fail-fast: they will throw java.util.ConcurrentModificationException if the underlying Set is modified, so be prepared for this if you expect to share selectors and/or key sets among threads. You're allowed to modify the selection key set directly, but be aware that you could clobber some other thread's Iterator by doing so.

If there is any question of multiple threads accessing the key sets of a selector concurrently, you must take steps to properly synchronize access. When performing a selection operation, selectors synchronize on the Selector object, the registered key set, and the selected key set objects, in that order. They also synchronize on the cancelled key set during Steps 1 and 3 of the selection process (when it deregisters channels associated with cancelled keys).

In a multithread scenario, if you need to make changes to any of the key sets, either directly or as a side effect of another operation, you should first synchronize on the same objects, in the same order. The locking order is vitally important. If competing

threads do not request the same locks in the same order, there is a potential for deadlock. If you are certain that no other threads will be accessing the selector at the same time, then synchronization is not necessary.

The `close()` method of `Selector` synchronizes in the same way as `select()`, so there is a potential for blocking there. A thread calling `close()` will block until an in-progress selection is complete or the thread doing the selection goes to sleep. In the latter case, the selecting thread will awaken as soon as the closing thread acquires the locks and closes the selector (see the section "Stopping the Selection Process").

Asynchronous Closability

It's possible to close a channel or cancel a selection key at any time. Unless you take steps to synchronize, the states of the keys and associated channels could change unexpectedly. The presence of a key in a particular key set does not guarantee that the key is still valid or that its associated channel is still open.

Closing channels should not be a time-consuming operation. The designers of NIO specifically wanted to prevent the possibility of a thread closing a channel being blocked in an indefinite wait if the channel is involved in a select operation. When a channel is closed, its associated keys are cancelled. This does not directly affect an in-process `select()`, but it does mean that a selection key that was valid when you called `select()` could be invalid upon return. You should always use the selected key set returned by the selector's `selectedKeys()` method; do not maintain your own set of keys. Understanding the selection process as outlined in the section "The Selection Process" is important to avoid running into trouble.

Refer to the section "Stopping the Selection Process" for the details of how a thread can be awakened when blocked in `select()`.

If you attempt to use a key that's been invalidated, a `CancelledKeyException` will be thrown by most methods. You can, however, safely retrieve the channel handle from a cancelled key. If the channel has also been closed, attempting to use it will yield a `ClosedChannelException` in most cases.

Selection Scaling

I've mentioned several times that selectors make it easy for a single thread to multiplex large numbers of selectable channels. Using one thread to service all the channels reduces complexity and can potentially boost performance by eliminating the overhead of managing many threads. But is it a good idea to use just one thread to service all selectable channels? As always, it depends.

It could be argued that on a single CPU system it's a good idea because only one thread can be running at a time anyway. By eliminating the overhead of context

switching between threads, total throughput could be higher. But what about a multi-CPU system? On a system with *n* CPUs, *n*–1 could be idling while the single thread trundles along servicing each channel sequentially.

Or what about the case in which different channels require different classes of service? Suppose an application logs information from a large number of distributed sensors. Any given sensor could wait several seconds while the servicing thread iterates through each ready channel. This is OK if response time is not critical. But higher-priority connections (such as operator commands) would have to wait in the queue as well if only one thread services all channels. Every application's requirements are different. The solutions you apply are affected by what you're trying to accomplish.

For the first scenario, in which you want to bring more threads into play to service channels, resist the urge to use multiple selectors. Performing readiness selection on large numbers of channels is not expensive; most of the work is done by the underlying operating system. Maintaining multiple selectors and randomly assigning channels to one of them is not a satisfactory solution to this problem. It simply makes smaller versions of the same scenario.

A better approach is to use one selector for all selectable channels and delegate the servicing of ready channels to other threads. You have a single point to monitor channel readiness and a decoupled pool of worker threads to handle the incoming data. The thread pool size can be tuned (or tune itself, dynamically) according to deployment conditions. Management of selectable channels remains simple, and simple is good.

The second scenario, in which some channels demand greater responsiveness than others, can be addressed by using two selectors: one for the command connections and another for the normal connections. But this scenario can be easily addressed in much the same way as the first. Rather than dispatching all ready channels to the same thread pool, channels can be handed off to different classes of worker threads according to function. There may be a logging thread pool, a command/control pool, a status request pool, etc.

The code in Example 4-2 is an extension of the generic selection loop code in Example 4-1. It overrides the `readDataFromSocket()` method and uses a thread pool to service channels with data to read. Rather than reading the data synchronously in the main thread, this version passes the `SelectionKey` object to a worker thread for servicing.

Example 4-2. Servicing channels with a thread pool

```
package com.ronsoft.books.nio.channels;

import java.nio.ByteBuffer;
import java.nio.channels.SocketChannel;
import java.nio.channels.SelectionKey;
```

Example 4-2. Servicing channels with a thread pool (continued)

```
import java.util.List;
import java.util.LinkedList;
import java.io.IOException;

/**
 * Specialization of the SelectSockets class which uses a thread pool
 * to service channels.  The thread pool is an ad-hoc implementation
 * quicky lashed togther in a few hours for demonstration purposes.
 * It's definitely not production quality.
 *
 * @author Ron Hitchens (ron@ronsoft.com)
 */
public class SelectSocketsThreadPool extends SelectSockets
{
    private static final int MAX_THREADS = 5;

    private ThreadPool pool = new ThreadPool (MAX_THREADS);

    // -------------------------------------------------------------

    public static void main (String [] argv)
        throws Exception
    {
        new SelectSocketsThreadPool( ).go (argv);
    }

    // -------------------------------------------------------------

    /**
     * Sample data handler method for a channel with data ready to read.
     * This method is invoked from the go( ) method in the parent class.
     * This handler delegates to a worker thread in a thread pool to
     * service the channel, then returns immediately.
     * @param key A SelectionKey object representing a channel
     *  determined by the selector to be ready for reading.  If the
     *  channel returns an EOF condition, it is closed here, which
     *  automatically invalidates the associated key.  The selector
     *  will then de-register the channel on the next select call.
     */
    protected void readDataFromSocket (SelectionKey key)
        throws Exception
    {
        WorkerThread worker = pool.getWorker( );

        if (worker == null) {
            // No threads available. Do nothing. The selection
            // loop will keep calling this method until a
            // thread becomes available.  This design could
            // be improved.
            return;
        }
```

Example 4-2. Servicing channels with a thread pool (continued)

```
        // Invoking this wakes up the worker thread, then returns
        worker.serviceChannel (key);
    }

    // ---------------------------------------------------------------

    /**
     * A very simple thread pool class.  The pool size is set at
     * construction time and remains fixed.  Threads are cycled
     * through a FIFO idle queue.
     */
    private class ThreadPool
    {
        List idle = new LinkedList( );

        ThreadPool (int poolSize)
        {
            // Fill up the pool with worker threads
            for (int i = 0; i < poolSize; i++) {
                WorkerThread thread = new WorkerThread (this);

                // Set thread name for debugging. Start it.
                thread.setName ("Worker" + (i + 1));
                thread.start( );

                idle.add (thread);
            }
        }

        /**
         * Find an idle worker thread, if any.  Could return null.
         */
        WorkerThread getWorker( )
        {
            WorkerThread worker = null;

            synchronized (idle) {
                if (idle.size( ) > 0) {
                    worker = (WorkerThread) idle.remove (0);
                }
            }

            return (worker);
        }

        /**
         * Called by the worker thread to return itself to the
         * idle pool.
         */
        void returnWorker (WorkerThread worker)
        {
            synchronized (idle) {
```

Example 4-2. Servicing channels with a thread pool (continued)

```
                idle.add (worker);
            }
        }
    }

    /**
     * A worker thread class which can drain channels and echo-back
     * the input.  Each instance is constructed with a reference to
     * the owning thread pool object. When started, the thread loops
     * forever waiting to be awakened to service the channel associated
     * with a SelectionKey object.
     * The worker is tasked by calling its serviceChannel( ) method
     * with a SelectionKey object.  The serviceChannel( ) method stores
     * the key reference in the thread object then calls notify( )
     * to wake it up.  When the channel has been drained, the worker
     * thread returns itself to its parent pool.
     */
    private class WorkerThread extends Thread
    {
        private ByteBuffer buffer = ByteBuffer.allocate (1024);
        private ThreadPool pool;
        private SelectionKey key;

        WorkerThread (ThreadPool pool)
        {
            this.pool = pool;
        }

        // Loop forever waiting for work to do
        public synchronized void run( )
        {
            System.out.println (this.getName( ) + " is ready");

            while (true) {
                try {
                    // Sleep and release object lock
                    this.wait( );
                } catch (InterruptedException e) {
                    e.printStackTrace( );
                    // Clear interrupt status
                    this.interrupted( );
                }

                if (key == null) {
                    continue;    // just in case
                }

                System.out.println (this.getName( )
                    + " has been awakened");

                try {
                    drainChannel (key);
```

Example 4-2. Servicing channels with a thread pool (continued)

```
            } catch (Exception e) {
                System.out.println ("Caught '"
                    + e + "' closing channel");

                // Close channel and nudge selector
                try {
                    key.channel().close( );
                } catch (IOException ex) {
                    ex.printStackTrace( );
                }

                key.selector().wakeup( );
            }

            key = null;

            // Done. Ready for more. Return to pool
            this.pool.returnWorker (this);
        }
    }

    /**
     * Called to initiate a unit of work by this worker thread
     * on the provided SelectionKey object.  This method is
     * synchronized, as is the run( ) method, so only one key
     * can be serviced at a given time.
     * Before waking the worker thread, and before returning
     * to the main selection loop, this key's interest set is
     * updated to remove OP_READ.  This will cause the selector
     * to ignore read-readiness for this channel while the
     * worker thread is servicing it.
     */
    synchronized void serviceChannel (SelectionKey key)
    {
        this.key = key;

        key.interestOps (key.interestOps( ) & (~SelectionKey.OP_READ));

        this.notify( );        // Awaken the thread
    }

    /**
     * The actual code which drains the channel associated with
     * the given key.  This method assumes the key has been
     * modified prior to invocation to turn off selection
     * interest in OP_READ.  When this method completes it
     * re-enables OP_READ and calls wakeup( ) on the selector
     * so the selector will resume watching this channel.
     */
    void drainChannel (SelectionKey key)
        throws Exception
    {
```

Example 4-2. Servicing channels with a thread pool (continued)

```
            SocketChannel channel = (SocketChannel) key.channel( );
            int count;

            buffer.clear( );            // Empty buffer

            // Loop while data is available; channel is nonblocking
            while ((count = channel.read (buffer)) > 0) {
                buffer.flip( );         // make buffer readable

                // Send the data; may not go all at once
                while (buffer.hasRemaining( )) {
                    channel.write (buffer);
                }
                // WARNING: the above loop is evil.
                // See comments in superclass.

                buffer.clear( );        // Empty buffer
            }

            if (count < 0) {
                // Close channel on EOF; invalidates the key
                channel.close( );
                return;
            }

            // Resume interest in OP_READ
            key.interestOps (key.interestOps( ) | SelectionKey.OP_READ);

            // Cycle the selector so this key is active again
            key.selector().wakeup( );
        }
    }
}
```

Because the thread doing the selection will loop back and call `select( )` again almost immediately, the interest set in the key is modified to remove interest in read-readiness. This prevents the selector from repeatedly invoking `readDataFromSocket( )` (because the channel will remain ready to read until the worker thread can drain the data from it). When a worker thread has finished servicing the channel, it will again update the key's interest set to reassert an interest in read-readiness. It also does an explicit `wakeup( )` on the selector. If the main thread is blocked in `select( )`, this causes it to resume. The selection loop will then cycle (possibly doing nothing) and reenter `select( )` with the updated key.

Summary

In this chapter, we covered *the* most powerful aspect of NIO. Readiness selection is essential to large-scale, high-volume server-side applications. The addition of this

capability to the Java platform means that enterprise-class Java applications can now slug it out toe-to-toe with comparable applications written in any language. The key concepts covered in this chapter were:

Selector classes
: The `Selector`, `SelectableChannel`, and `SelectionKey` classes form the triumvirate that makes readiness selection possible on the Java platform. In the section "Selector Basics," we saw how these classes relate to each other and what they represent.

Selection keys
: In the section "Using Selection Keys," we learned more about selection keys and how they are used. The `SelectionKey` class encapsulates the relationship between a `SelectableChannel` object and a `Selector` with which it's registered.

Selectors
: Selection requests that the operating system determine which channels registered with a given selector are ready for I/O operation(s) of interest. We learned about the selection process in the section "Using Selectors" and how to manage the key set returned from a call to `select()`. We also discussed some of the concurrency issues relating to selection.

Asynchronous closability
: Issues relating to closing selectors and channels asynchronously were touched on in the section "Selector Basics."

Multithreading
: In the section "Selection Scaling," we discussed how multiple threads can be put to work to service selectable channels without resorting to multiple `Selector` objects.

Selectors hold great promise for Java server applications. As this powerful new capability is integrated into commercial application servers, server-side applications will gain even greater scalability, reliability, and responsiveness.

OK, we've completed the main tour of `java.nio` and its subpackages. But don't put your camera away. We have a couple of bonus highlights thrown in at no extra charge. Watch your step reboarding the bus. Next stop is the enchanted land of regular expressions.