Scalability is nearly the top issue of every corporate Web site, for both intranets and extranets. How do the companies scale to a growth in traffic? Corporations constantly need to be prepared for growth because their business plans dictate growth over time. The concerns are, How is a surge in traffic handled as business grows? Do systems need to be rewritten? Do bigger computers with more horsepower need to be deployed? Will current systems built on current technology scale to accommodate new growth?

Sometimes growth comes a little too unexpectedly. In July of 2000, for example, Qwest Communications had approximately 10,000 employees. Qwest bought out US West, which had 67,000 employees. Qwest’s systems needed to scale up from handling 10,000 to 77,000 employees overnight. In terms of the Internet, Qwest’s Web sites went from handling only network access customers to handling a 13-state local telephone region. Use of Qwest’s Web sites grew from thousands of hits per day to hundreds of thousands of hits per day. Scalability was a critical issue for this organization.

If you are reading this book, you must be about to develop your Web systems in Java, or perhaps you already have. You have made a very good decision because Java runs on many different kinds of platforms, so hardware options help on one side of the scalability equation. The other side of the equation is software—that is, building applications that scale well. More horsepower takes scalability only so far. Designing the software correctly can be even more critical to scalability.

The key to building scalability into an application is object reuse. By object reuse we don’t mean taking an object-oriented approach to development, but rather reusing resources such as database connections and memory buffers. We call this pooling resources. Through pooling, we can create a specific number
of resources, borrow resources when we need them, and put them back into the pool when we have finished with them, allowing another process or user to use them.

In this chapter we will examine the concept of pooling and why it is important to scalability. We will build a base pool object that allows us to reuse all kinds of resources. We will then look at how to manage pools and how pools fit into the enterprise servlet framework.

**What Is a Pool?**

When we talk about pools with respect to Web development, we are not talking about relaxing in a warm, small body of water, sipping a margarita, and getting a nice tan. We are talking about pooling together resources to be “borrowed” by different processes. A pool is a readily available supply—a finite supply from which resources are borrowed, used, and returned for reuse. To clarify, let’s look at a familiar real-world example of a pool.

Most of us have been to a video store at one time or another. Imagine that your local video store, Freddy’s Video Rentals, ordered five copies of a particular video, thus creating a pool of videos. The owner, Freddy, figured that by having five copies of a particular video on hand, he would be able to keep his customers happy because almost never are five copies of the same movie checked out at once. You go to Freddy’s to check out this video, and you notice that two videos are left. After you check out the movie, only one copy is left. But during the transaction, your friend Jerry walks through the door to return the same movie that you’re checking out. “Hey Jerry, how did you like the movie?” you ask. He says, “It was great. I’m just returning it now.” Now again two movies are left. You go home, watch the movie, and return it the next day. As you return the movie, you notice that there are three movies on the shelf, so with the addition of your returned copy, four copies are available. Then you spy your friend Pete, who is standing in line with the same movie. You call out, “Hey Pete! Awesome movie!” Pete says, “Yeah, I’m going to watch it tonight. I hear it’s good.” Pete has the fifth movie.

Freddy purchased five copies of the same movie, creating a pool of that particular movie. His customers (or users) check out and borrow the movie as illustrated in Figure 7.1. After watching the movie, each customer returns the video to the pool so that another customer can use it. Freddy has done an excellent job of pooling his resources to fit his customer’s needs.

Would it have been prudent for Freddy to purchase a new video for each of his customers, then let them check out the video and return it? Absolutely not. Such an approach would have been too expensive, and Freddy would have
gone out of business. Freddy gauged his customer traffic and was able to pool
the right number of videos so that all of his customers were happy.

Now let’s see where scalability fits into this scenario. Say that the year’s
Academy Award Best Picture comes out on video. Although the movie has
received rave reviews, many people have been very busy and haven’t had the
chance to see it in the movie theaters. Others have seen the movie and liked it
so much that they can’t wait to see it on video. Freddy knows that this movie
will be a hit, so he orders 10 copies of it for his store, instead of the usual 5.
But Freddy hasn’t anticipated just how popular the movie will be; all 10 copies
are checked out in the first ten minutes after the shipment is received from the
distribution company. In addition, 20 people are waiting for the video. Freddy
needs to scale his business to meet the demand of his customers.

Freddy places a rush order for 20 more videos. When they arrive, he has a
total of 30 copies of the movie. The next day, 2 copies are left on the shelf.
Freddy has scaled his business very well for the increased growth in demand for
the video. What happens when the fury is over and Freddy has a lot of videos
taking up space on his shelf? He has a fire sale and sells the excess videos at a
great discount!

Web development is not so different from the video store scenario. Just as
Freddy would not order a new video for each of his customers, we would not
want to assign a new database connection to each user going to a Web site.
With thousands of users hitting a Web site, the database would not be able to
handle all the concurrent connections. Just as thousands of videos are too
expensive for Freddy, thousands of database connections are too expensive for
us, but in this case in terms of memory resources and CPU usage.
We can scale resources just as Freddy scales his business with videos. If our Web site is hit by only a few people at any given moment, we need only a few database connections. If the Web site is high capacity and is hit by as many as 30 users at any given moment, we should be prepared to have a pool of 30 database connections. If customers have to wait a long time for a connection, they will complain about the performance of the Web application. So a good approach is to pool a number of resources that matches the largest spike in hits to the Web server.

The advantages of pools do not stop at scalability. In Java, continual creation and destruction of objects are bad things. These operations increase the rate at which the garbage collector executes, and for a high-capacity Web site, they could freeze a Web server for a period of time. An application with less destruction and more reuse of objects will manage memory better and run more efficiently. Scalability, object reuse, and refined memory management are all good side effects of object pooling.

Using Pools in Web Development

What kinds of resources do we pool when we build applications? Just about any kind of resource can be pooled. If you have a CD-ROM farm, for example, you can pool each of the CD-ROM drives to serve data. As in the Web example just described, database connections are one of the most commonly pooled resources. Large memory blocks are also an excellent candidate to pool as a resource.

So how do we pool objects? An object pool is a list or group of objects, usually stored in an array. The array keeps track of each object’s state—either in use and unavailable, or available for use. The array also provides the methods to allow the child threads to “check out” and “check in” the objects. Such a pool is not very different from the pool of videos in a video store. Typically, a developer or administrator sets the number of objects that the pool will contain. The pool creates these objects at startup of the application or server. As each thread, process, or user requires a resource from the pool, it calls a checkout method on the pool. When the pool receives this call, it moves a pointer along the object array and looks for an object that is available. The pool then marks the object as unavailable and passes it back to the calling thread. The thread uses the object as it needs; then it calls the checkin method and passes the object back to the pool. The pool finally takes the object and marks it as available so that may be used by other threads.

When all objects are checked out, the threads wait and retrieve the first available object on a first come, first served basis. This is very similar to how the video store pool works. The video store has a certain number of videos.
Each person checks out a video, watches it, and checks it back in after using it. If the number of people who want to watch the video exceeds the number of available videos, some people must wait until a copy is checked back in. Most video stores have a waiting list that is on a first come, first served basis, so they let the first customer on the list know when a copy of the video becomes available.

Just as in the video store, our pool needs to be scalable. In the earlier video store example, for instance, Freddy purchased more copies of a particularly popular movie so that nobody would have to wait for it. We should do the same when we use pools. Ultimately we want just enough objects in the pool that we maximize the resources without being wasteful and without making any thread wait.

When threads wait, end users wait. The results are slow sites and angry customers. With scalability, we can grow the system to meet the demands of users. Therefore, we need to choose a pool size that fits the users’ needs and increase the size of the pool as the needs increase.

There are two different ways of doing this. The first is the manual method: We choose a pool size based on our expectations of the users’ demands. Then we monitor the peaks and valleys of the users’ demands and size the pool accordingly. As users’ demands grow, we increase the pool size. The second method is dynamic pool growth. We build a pool that constantly grows according to the needs of the users. If the pool objects are all checked out, the next user who comes along is given a brand-new object as the pool automatically grows by one.

Although the second method may appear better because it requires no monitoring and the pool is responsible for its own scaling, it has some disadvantages. An outsider who knows that objects are being dynamically created in the application could maliciously attack a Web site by making hundreds of calls to the application at once. This tactic could force the servers to run out of memory very quickly as the pool continually added objects to itself in an attempt to scale to the load. The disadvantage of the first method is that it requires constant monitoring of the pools to be sure users are not waiting. So each method has its advantages and disadvantages for scaling pools.

Object reuse seems like a great concept, but is there a time when we do not want to use a pool? Absolutely! Pools do not solve all problems when we’re building Web applications. One instance in which a pool should not be used is for a long-running process. Using an object that takes ten minutes to run a process, such as a large database query, could be detrimental to a site. With many users, and thus many of these long-running processes running at the same time, the pool could be used up very quickly. As a result, the Web server would appear to hang, or we might be forced to send an error message to the user.
stating that all connections were in use and to try again later. Just as the video store might lose customers who are not willing to wait a long time for a video to come back and thus go to another video store, so might we lose users of our Web sites. So one rule of thumb in deciding whether an object is a good candidate for a pool is that should not belong to a long-running process.

The Base Pool Object

Let’s look more closely at a pool and what it will take to build a pooling object. From what we understand of pools so far, it’s clear that each pool minimally needs an object array that contains flags to describe each object’s availability. It needs methods to create and initialize the objects, and methods to check in and check out the objects. But this is a lot of code to build for each object. Extend that idea to an application that has two database pools and three memory pools, and the amount of code required becomes unmanageable.

The answer to this problem is to create a base pool object that can be extended to create any sort of object pool. This base pool object would allow us to create a very minimal amount of code when developing an object pool. In September 1999, while looking to build such a base pool object. I read an article\(^1\) about creating a base pool object for a graphics pool for the Java Abstract Windows Toolkit (AWT). This article gave me an idea about building a generic base pool that can be used for any kind of object. The key requirements for any pool are that it must

1. Maintain each of the object’s states—through a generic object array.
2. Be able to create its own objects.
3. Have means for checking objects out and back in.
4. Be able to perform operations just before an object is checked out or in (such as initializing an object before it is checked out and cleaning it up before it is checked in).
5. Be able to replace an object with a newly created object. This capability could be helpful if the object became invalidated for some reason and the pooled object needed to be re-created. One such scenario would be an invalidated database connection, which would mean that we would have to re-create the connection on the fly and replace the old with the new.

Let’s begin developing a base pool object with the interface that it may implement. The base pool needs to allow object pools to perform operations on

---

an object before it is checked out and before it is checked back in. For example, if we have a memory object that needs to be initialized or to have its internal data set to particular values before it is used, the framework needs to allow the pool to perform these actions on the object. The same capability may be needed just before the object is checked back in. For instance, if we have a pool of StringBuffer memory areas, we might wish to have the framework automatically reset the size of StringBuffer and empty it before checking it in.

We do this by creating an interface with the beforeCheckin() and the beforeCheckout() methods. By creating an interface, we can create default implementations of these methods in the base pool. That way the developer's pool does not need to create its own versions if no operations have to be performed on the object before checkin or checkout. In other words, if the developer decides to override these methods, the framework will call the developer's version instead of the default version. Let's call this interface PoolInterface. Listing 7.1 shows the code for PoolInterface.

**Listing 7.1** The PoolInterface class

```java
package enterprise.common;

public interface PoolInterface {
    public void beforeCheckin(Object object);
    public Object beforeCheckout(Object object) throws Exception;
}
```

A technical description of the various methods of PoolInterface is necessary because it is important to understand what's going on here. The beforeCheckout() method takes an object (Object) as a parameter. The object that is passed to this method is the same one that was checked out by the framework. But we want this method to return Object as the return type. Why? Why not just claim the return type as void and use the parameter referenced Object? Requirement number 5 for pools states that we may need to re-create an object, so the object reference could change. By passing Object as a parameter, the reference point to the framework remains the same, so there would be no way to return a new object to the user unless we return Object as a return type.

Why would we need to re-create an object? A database connection is a perfect example. If the database administrator needs to restart the database, all of the connections in the database will become invalid. A beforeCheckout() method could detect a bad connection, re-create the Connection object, and internally replace the old one with the new. Because Connection is passed as a parameter to beforeCheckout(), we must return the new Connection object to...
be used. Only by having the beforeCheckout() method return an Object type can we do this.

The beforeCheckin() method is a little simpler. It does not need to have a return type, so we can declare it as void. Because we are checking in an object, the framework will need to use this object’s reference to find its place in the array. We cannot simply change the object at this point, or the reference will change and we will not be able to check in the object. Besides, we are done with the object, so we really aren’t interested in creating new versions. With this method we just want to clean up the object.

Now we are ready to develop the base pool object. Because this is a generic pool, we can simply call this class Pool. The Pool class will be declared abstract for two reasons. First and foremost it is a base pool. Without an implementation of code that pools some kind of an object, this class is completely unusable. Therefore, by declaring the Pool class abstract, we ensure that it cannot be used directly. Rather, this class must be extended as a part of an object to create the pool. The second reason that the Pool class must be abstract is that we plan to force the declaration of some methods in a subclass. By declaring the class abstract and declaring abstract methods, we can ensure that subclasses implement these abstract methods.

The whole concept behind a pool is to manage an array of objects. We manage this array by setting a flag upon checkin and checkout of an object to state whether the object is currently available or not. Each stored object must have one of these flags (which we will call isUsed). The easiest way to implement a one-to-one correlation between the flag and the object is to create a nested class that contains the isUsed flag and Object as member variables. We will call the private class PoolObject, thereby forcing a one-to-one correlation between the flag and the object when we create an array of PoolObject objects.

```java
package enterprise.common;

public abstract class Pool implements PoolInterface
{

    private class PoolObject
    {
        public boolean isUsed;
        public Object object;

        public void setObject(Object o)
        {
            object = o;
        }
    }

    private PoolObject objectArray[];
```
We will declare a couple of constants to use within the class. These will be described when we get to the code that uses them.

```java
private static final int SLEEP_MILLISECONDS = 100;
private static final int MAX_NUMBER_OF_TRIES = 10;
```

The following are some member variables that are used in managing the pool. The number of objects in the internal array will be kept in the capacity variable. The `indexPointer` variable will contain the index element in the array that identifies the next object in the array that will be checked out.

```java
private int capacity;
private int indexPointer;
```

Next we will create the abstract method that all pools that extend the `Pool` class must implement. All pools must somehow create objects for the pool. Therefore, each pool should implement a `createObject()` method that returns an object. We want the framework to call the `createObject()` method for each object that will be placed into the array. By making this method abstract, we can force the inherited classes to include it and guarantee that the version in the subclass will be executed when the pool is being created. We will see how this works when we get to the pool initialization code. For now, let’s declare an abstract class for `createObject()`.

```java
public abstract Object createObject() throws Exception;
```

The default constructor will be declared protected and is used only when we want an inherited class to create its own constructor. This protection will allow certain inherited classes to compile.

```java
protected Pool()
{
}
```

The directly usable constructor takes a parameter for the size of the pool. The constructor then calls the internal `createPool()` method.

```java
public Pool(int maxSize) throws Exception
{
    createPool(maxSize);
}
```

The `createPool()` method is what creates the object array. We first want to be sure that we have a pool size of at least one. We do this because a pool size with no objects is no good. Such a pool would be completely unusable, so we require the pool size to be 1 or greater, or an exception is thrown. The first thing we do is create a `PoolObject` array of the requested pool size. Then we
create the objects. This is where we call the abstract createObject() method 
mentioned earlier. Because it is abstract, it actually calls the version of 
createObject() that is in the inherited class. As you can see, for each element 
in the array, createObject() is called to retrieve a new object and place it in 
this array.

protected void createPool(int maxCapacity) throws Exception
{
    if (maxCapacity < 1)
    {
        throw new Exception("maxCapacity is less than 1.");
    }

    // Create the pool
    this.capacity = maxCapacity;
    objectArray = new PoolObject[capacity];

    // Create the objects
    for (int i = 0; i < capacity; i++)
    {
        objectArray[i] = new PoolObject();
        objectArray[i].isUsed = false;
        objectArray[i].object = createObject();
    }
}

We will also create some helper functions that return the array size (the 
capacity) and a method that allows the developer to free all of the pool 
connections.

public synchronized int getCapacity()
{
    return (capacity);
}

public synchronized void openAll()
{
    int i = 0;
    for (i = 0; i < capacity; ++i)
    {
        objectArray[i].isUsed = false;
    }
}

Here we will declare the default implementations of beforeCheckin() and 
beforeCheckout() methods as required by PoolInterface. These methods will 
be called if the inherited pool class does not declare its own versions.

public void beforeCheckin(Object object){};

public Object beforeCheckout(Object object) throws Exception
{
    return object;
}
Now let’s develop the real meat of the base pool: the checkin and checkout methods. These methods will be synchronized because they are publicly accessible and only one thread at a time should be allowed access to the object array. This prevents the possibility of multiple requests at the same time to retrieve the same object.

Let’s start with checkout because it is more complex than checkin. We’ll call this method `checkOutPoolObject()`, and it will be responsible for going through the array in a round-robin fashion until it finds an available object (in other words, an object whose `isUsed` flag is set to `false`). It works by incrementing the `indexPointer` flag and checking the `PoolObject.isUsed` variable. Whenever it finds a `PoolObject` element in the array whose `isUsed` flag is set to `false`, `checkOutPoolObject()` knows that that object is available and it sets the `isUsed` flag to `true` (making the object unavailable), calls the `beforeCheckout()` method on the object, and returns the resulting object to the user.

However, if `indexPointer` has gone through the entire array a full time, the thread goes to sleep for the amount of time specified by the constant `SLEEP_MILLISECONDS`, and then wakes up and tries again. During the sleep period, it is hoped that another thread will check in an object, and the original thread will then have access to this available object. If not, the original thread will continue this process for the number of tries specified by another constant defined in this class, `MAX_NUMBER_OF_TRIES`. If the value specified by `MAX_NUMBER_OF_TRIES` has been reached, the exception `PoolObjectNotAvailableException` is thrown.

The code for `PoolObjectNotAvailableException` is shown in Listing 7.2 on page 253. In a nutshell, when we want to check out an object and none are available, we try as many times as specified by `MAX_NUMBER_OF_TRIES` and sleep for the amount of time specified by `SLEEP_MILLISECONDS`. If we cannot get an object, `checkOutPoolObject()` throws the `PoolObjectNotAvailableException` exception. If we do get an object, it executes the `beforeCheckout()` method on the object and returns the resulting object.

```java
public synchronized Object checkOutPoolObject() throws Exception {
    Object object = null;
    int tryCount = 0;
    int elementCounter = 0;

    while (tryCount < MAX_NUMBER_OF_TRIES) {
        // We are about to go through all the array elements once, so
        // we will set elementCounter = 0
        // ...
elementCounter = 0;
while( elementCounter < capacity)
{
    // Find an object that is not being used
    if (!objectArray[indexPointer].isUsed)
    {
        // Found one! Now let's reserve it.
        object = objectArray[indexPointer].object;
        objectArray[indexPointer].isUsed = true;
    }

    // Let's increment the index pointer
    indexPointer++;

    // Wrap the pointer around if we reached the top
    if (indexPointer == capacity)
        indexPointer = 0;

    // Increment the number of elements we have looked at
    elementCounter++;

    // If we have an object, exit the element loop
    if (object != null)
    {
        break;
    }
}

// Did we get an object?
if (object == null)
{
    // No, so let's increment the try count and sleep
    // to allow someone to check in a value
    tryCount++;
    try
    {
        wait(SLEEP_MILLISECONDS);
    }
    catch (java.lang.InterruptedIOException ie)
    {
    }
    break;
}
else
{
    // We have an object, so get out of the loop
    break;
}

// If we couldn't get an object after trying
// MAX_NUMBER_OF_TRIES,
// then we will throw an exception
if (object == null)
{
    throw new PoolObjectNotAvailableException(
        "Cannot get object from pool.");
}

return beforeCheckout(object);

Our checkin method, checkInPoolObject(), takes the object that we want
to check in as a parameter. When we check in the object, we look for its place
in the array. When we find its place, we call the beforeCheckin() method with
the object passed as a parameter to perform any custom cleanup actions on the
object, if this method is defined in the inherited pool class. Ultimately, then, we
check in the object by setting the associated isUsed variable to false. If
the object is not found in the array, the method throws the exception
IllegalArgumentException.

public synchronized void checkInPoolObject( Object object )
    throws IllegalArgumentException
{
    boolean found = false;
    for (int i=0; i < capacity; ++i)
    {
        if (objectArray[i].object == object)
        {
            beforeCheckin(object);

            objectArray[i].isUsed = false;
            found = true;
            break;
        }
    }

    if (! found)
    {
        throw new IllegalArgumentException(
            "Object does not belong to pool");
    }
}

Finally, we need a method that allows us to replace an object in the pool
array. This method will take two parameters—the old object and the
new object—and it works very similarly to how checkInPoolObject() works.
It looks in the array for the old object. If it finds it, it replaces the
PoolObject.object reference with the new object. We make this a public syn-
chronized method because once again we are potentially altering the array. It is
public synchronized void replacePoolObject( Object oldObject, 
Object newObject)
throws IllegalArgumentException
{
    boolean found = false;
    for (int i=0; i < capacity; ++i)
    {
        if (objectArray[i].object == oldObject)
        {
            objectArray[i].object = newObject;
            found = true;
            break;
        }
    }
    if (! found)
    {
        throw new IllegalArgumentException(
                        "Object does not belong to pool");
    }
}

As stated earlier in the chapter, if we try to check out an object and after 
MAX_NUMBER_OF_TRIES no such object is available to be retrieved, we will receive 
the exception PoolObjectNotAvailableException. Listing 7.2 shows the code 
for PoolObjectNotAvailableException.

Listing 7.2  The PoolObjectNotAvailableException class

package enterprise.common;

public class PoolObjectNotAvailableException extends Exception
{
    public PoolObjectNotAvailableException()
    {
        super();
    }

    public PoolObjectNotAvailableException(String msg)
    {
        super(msg);
    }
}
Using the Pool Object

Now that we have a base pool object (Pool), how do we use it? Before creating a pool, we need an object that will be pooled. Let's build a simple object that may be useful in building Web pages. On a high-capacity Web site that uses a lot of rich and robust templates, we would create and destroy many StringBuffer objects. Although it is much better to use StringBuffer objects than to use String objects, over time the hunt for memory during the dynamic building of large instances of StringBuffer could create havoc with the garbage collector.

Default initialization of a StringBuffer object allocates a size of 16 characters. A typical template in an application may be approximately 4,000 characters (4K). If we create a StringBuffer object and build a template in it, the virtual machine needs to allocate an additional 3,984 characters when building the template. Each time we call the append() method on the StringBuffer object, the virtual machine needs to find space to allocate room for the buffer piece by piece.

On a high-capacity site, many threads may be doing this at the same time—creating a workload that can heavily fragment memory and keep the garbage collector busy. If we can pool StringBuffer objects and preallocate the buffer size, we can reuse these objects. This approach prevents memory fragmentation, and the garbage collector practically does not need to execute because no objects are being destroyed. Let's create an object called SBObject as a wrapper class for StringBuffer. This object will preallocate 4,000 characters for StringBuffer. We will include an int variable that contains an identifier for the buffer when we create the pool. Listing 7.3 shows the code for SBObject.

Listing 7.3 The SBObject class as a StringBuffer object

```java
import java.util.*;

public class SBObject {
    public StringBuffer sb = new StringBuffer(4000);
    public int bufferId = 0;
}
```

To pool all the SBObject objects, we start by extending the Pool object. Only a couple of rules apply to creating a pool in this scenario. The first is that the constructor must call the super() method with an int parameter for the size of the pool. We do this so that the superclass can create the object array and all of the objects. It's a good idea to create a constructor that takes an int variable as a parameter so that the pool size can be defined outside the object.
Second, because the Pool class is abstract, it forces the inherited class to implement the createObject() method, so the pool will need to create the objects in this class. The createObject() method will create a new instance of SBObject, which automatically allocates a size of 4,000 characters for its internal StringBuffer object, and will set the bufferId parameter to an integer that uniquely identifies the object. When we use the pool, bufferId will show us which object is being used.

These are the two required rules for creating a pool. The two optional rules are to create beforeCheckin() or beforeCheckout() methods. Remember that these methods are used when we need to perform actions on the object before checking in or checking out an object. In this example, we want to clean up an instance of SBObject and its internal StringBuffer object before checking it in. Therefore we will implement the beforeCheckin() method. In this method we will set the length of StringBuffer to zero. This does not deallocate the size, but instead makes the StringBuffer contents disappear, preparing the StringBuffer object for the next user. Because we are creating a pool of SBObject objects, let’s call the pool SBOPool. Listing 7.4 shows the code for SBOPool.

**Listing 7.4** The StringBuffer object pool, SBOPool

```java
import enterprise.common.*;

public class SBOPool extends Pool {
    private static int counter = 0;

    public SBOPool(int maxSize)
        throws Exception
    {
        super(maxSize);
    }

    public Object createObject() throws Exception
    {
        SBObject sbo = new SBObject();
        sbo.bufferId = ++counter;
        return sbo;
    }

    public void beforeCheckin(Object o)
    {
        SBObject sbo = (SBObject) o;
        sbo.sb.setLength(0);
    }
}
```
As Listing 7.4 shows, creating the pool is quite simple, but how do we use it? Where do we create it? How do we check objects out and check them back in? We usually initialize the pool when the application starts. This is a one-time action. Somewhere in the application’s initialization we would declare the following:

```java
SBOPool sboPool = new SBOPool(10);
```

This declaration creates the pool and ten objects in the pool’s object array. When we are ready to use an object, we will want to check it out and check it back in when we are finished. The following code would be used to retrieve and use the pooled objects:

```java
SBObject sbo = null;
try {
    sbo = (SBObject) sboPool.checkOutPoolObject();
    // Use the object
    . . .
} finally {
    // If we have an object, then we check it back in
    if (sbo != null) {
        sboPool.checkInPoolObject(sbo);
    }
}
```

This construct is required when we need a pooled object. What we are doing is checking out an object within a `try` block. We do this to guarantee that the object will be checked back in, no matter what happens. If an exception is thrown, or the user needs to exit the method, the `finally` clause is guaranteed to execute, so we can also guarantee that the object will be checked back in.

As you can see, creating a pool and using a pool are very simple with the `Pool` base class. Let’s see the pool work in an example.

### Using the Pool: An Example

In the previous section we created a `StringBuffer` object and an associated pool that preallocates a certain size for `StringBuffer`. Let’s implement a sample application to use this pool and object. The goal of this example is to show where we would create a pool and where we would use it. This example will use the `SBObject` and `SBOPool` classes that we created earlier. The reason we
created SBObject was to reuse the internal StringBuffer object for potentially large templates. So we will create an HTML template that allows us to show what pool we are using (now you see why we have a buffer identifier in the object).

Let’s name this application (and thus also the base dispatch servlet) PoolExample. The application will have one page and thus one dispatch class, which we will call PoolPage. We will use one template, which we will call pool.html. We’ll begin with the configuration file, as shown in Listing 7.5.

Listing 7.5  The configuration file for PoolExample

```
Application.Base.URL=/servlet/PoolExample
Default.Method=pool
Method.pool=PoolPage

# Place the default template path here
Default.Template.Path=c:\Book\Chapter 07\Example 1\templates
Template.pool=pool.html

# Place the log file path here
Log.File.Path=c:\Book\Chapter 07\Example 1\PoolExample.log
```

Now let’s build the base dispatch servlet, PoolExample. This servlet isn’t too different from the base servlets that we have developed in other chapters, except that we will create a static member as the pool object and initialize it in the `initializeApp()` method (see Listing 7.6). Normally, having accessible member variables in a servlet is not good, unless they are thread-safe through synchronization. Of course, our pools are thread-safe because all of the accessor methods that we use in the pool are synchronized. In this example, we will use a pool size of five objects.

Listing 7.6  The PoolExample main dispatch servlet with pool initialization

```
import enterprise.servlet.*;
import enterprise.common.*;
import java.io.*;

public class PoolExample extends BaseEnterpriseServlet
{
   public static SBOPool sboPool = null;

   public String getAppName()
   {
      return "PoolExample";
   }
```
public void initializeApp(AppContext appContext) throws Exception
{
    sboPool = new SBOPool(5);
}

public void destroyApp(AppContext ac)
{
}
}

PoolExample has one page, called PoolPage. This page will check out one instance of SBObject and use the pool.html template with that object's StringBuffer. The checkout and checkin code is used in the try/finally block shown earlier (at the end of the section titled Using the Pool Object). The substitution for the template will place SBObject.bufferId in the string, so you may see which pool object is being used for displaying the page. Listing 7.7 shows the code for PoolPage.

Listing 7.7 The PoolPage class

import enterprise.servlet.*;
import enterprise.html.*;
import java.io.*;

public class PoolPage extends HTTPMethod
{
    public void execute() throws Exception
    {
        SBObject sbo = null;

        try
        {
            PrintWriter out = m_response.getWriter();
            TemplateCacheList tcl = m_context.getTemplateCacheList();

            HTMLCompiledTemplate hct = tcl.get("pool");
            HTMLTemplate ht = hct.createHTMLTemplate();

            sbo = (SBObject) PoolExample.sboPool.checkOutPoolObject();

            ht.substitute("poolno", sbo.bufferId + "");
            ht.substitute("BASEURL", m_context.getAppBaseURL());
            ht.toStringBuffer(sbo.sb);
            out.println(sbo.sb.toString());
        }
        finally
        {
            if (sbo != null)
            {
                sboPool.checkInPoolObject(sbo);
            }
        }
    }
}
PoolExample.sboPool.checkInPoolObject(sbo);
}
}

Finally, we need the template—pool.html—that PoolPage will use. Listing 7.8 shows the code for pool.html.

Listing 7.8 The pool.html template

```html
<html>
<head>
<title>PoolExample</title>
</head>
<body bgcolor="#FFFFFF">
<center>
<h2>This HTML Page has been created using....</h2>
<br>
<br>
<h3><font color="#3333FF">StringBuffer Object Pool Number ${poolno}.</font></h3>
<br>
<br>
<form>
<input type="button" name="Button" value="Click here to get another Pool Object"
onClick="document.location='${BASEURL}';">
</form>
</center>
</body>
</html>
```

Now we are ready to set up and execute the application. The setup program registers the servlet and the ConfFile parameter in the servlet engine to point to the configuration file that we defined in Listing 7.5. When the application is being run, the output should be similar to what Figure 7.2 shows. Notice that we get the SBObject object in round-robin fashion. As we get the fifth object, notice also that the pool wraps around to the first one in the array.

Another idea is to have a friend test the servlet out with you on another browser. Try hitting the server several times at the same time as your friend. You will notice that the server still distributes objects in a round-robin fashion. As we get the fifth object, notice also that the pool wraps around to the first one in the array.

You can try with a few friends hitting the server at the same time. See if you can get the server to “wait” for an available connection. This may be difficult to do because the object is used for only a very short time and is checked in and out very rapidly. This test helps show the power of pooling heavily used and
large objects. Appreciation for a pool becomes really clear on a high-capacity site, where you will notice that the memory usage of your application remains static and the garbage collector does not have to work as hard to clean up after the objects.

The Pool Anomaly

The information about pools presented in the previous section is very valuable for the use of pooled resources. With this knowledge, however, comes a huge burden. The try/finally block that we must use when we check objects in and out must be present when a pooled object is being used. This is a critical requirement because we cannot guarantee that the object will be checked back in without it. The anomaly in this scenario is that the developer bears the responsibility of ensuring that the checkin is implemented in the object. We are placing trust in the developer that this code will be executed in every piece of code that uses a pooled object.

It is very easy to check out an object and forget to check it back in. Doing so may cause the Web servers to freeze up very fast as they wait for an object.

Figure 7.2 Output of the sample application
Many of the pool object APIs that are available today, including open-source APIs and APIs that are included as part of a vendor’s Web server, such as WebLogic, require this try/finally block to be implemented in code when a pooled object is being used. However, too often this code is missed during use of a pooled object and the Web server freezes. Generally both users and developers swear that the problem is not theirs and that the problem is the fault of the servlet engine vendor. After many attempts to find the problem, the culprit almost always turns out to be a piece of code that forgot to check in an object. Such coding errors are very difficult to find and debug.

One reason for implementing the constant MAX_NUMBER_OF_TRIES in the base Pool object is that it allows us to trap the condition when all of the objects have been used up. Receiving the PoolObjectNotAvailableException exception is a reasonably good indication that some of the code is not checking in objects. This exception helps prevent the appearance that the Web server is freezing up. However, the best scenario would be to avoid this situation altogether. If we rely on developers to execute the try/finally block, we risk running into code that does not comply with use of a pool. Is there something we can do to relieve the developer of this burden?

The answer is a resounding yes! In the enterprise servlet architecture, everything revolves around the base-servlet concept. By using a base servlet, we can ensure that pool management is handled in BaseEnterpriseServlet and make sure that any objects checked out of pools are checked back in. In other words, we need to have a centrally located pool manager that monitors which objects are checked out and ensures that the objects are checked back in. To monitor the pools, such a manager would need a list of all available pools. The pool manager will handle requests for objects to be checked out of each pool and then make sure those objects are checked back into their respective pools.

To understand this concept, let’s examine the video store example again. The pool manager would be the customer service person (CSP). A movie belongs to one pool of videos. A different movie belongs to another pool of videos. Therefore we have many pools of videos in the video store—one for each movie. When a customer wants a particular movie, the CSP literally checks out the video and hands it to the customer. A customer who forgets to bring the borrowed movie back to the video store after watching it usually gets a call from the CSP—a reminder that the video needs to be checked back in. The CSP is ensuring that the video is checked back into the pool so that it can be reused by others. When the video is returned, the CSP places it back in its respective pool on the shelf so that it can be checked out again. In the development world, the pool manager is analogous to the video store’s CSP, and the pool list is analogous to the list of movies. In the next section we will create objects to handle these two roles.
Case Study

The Marketing Department of a large corporation hired a very well-known consulting company to create an executive information system product. This product was a Web-based application that allowed the executives to drill down sales data at any given moment to analyze departmental sales quota and revenue bases.

The application had a polished interface and was the type that would be expected from this consulting firm. However, the Web server appeared to “freeze up” at random intervals. The only way to get the application running again was to restart the Web server. This became a real problem because the server was freezing up many times a day, and the executives were becoming concerned about the viability of the product.

The consulting company investigated and found no problems with its code. The consultants blamed the Web server and asked the Web server’s vendor to look into the problem.

After investigating, the vendor stated that there was no problem with the server. Frustrated, the Marketing Department asked IT’s crack Java team to analyze the problem.

The Java team interviewed the consultants and asked if they were using any pools. The consultants replied that they were using pools for connections to the database. The Java team had seen this kind of behavior before and instantly believed the culprit was that some code was not checking in objects.

Because there were hundreds of servlets in this application, the consultants had to comb through many lines of code. Four days later, they had found seven servlets that were not checking in code. The consulting company quickly made the change to the code and the server never froze up again.

This example shows that a lot of responsibility falls on the shoulders of the developer to be sure to check in pooled objects after using them. Having learned a lesson, the consulting company and the client’s Java team worked together to design a way for objects to be automatically checked back in after use.
**PoolList and PoolObjectManager**

Let's start with the pool list. We need an object that contains a list of all the available pools from which to check out objects for the application. We will call this object PoolList. This object is nothing more than a glorified Hashtable object. It will store the pools by name. For example, we may have three pools in one application. We would name the Oracle connection pool OraPool, the graphics pool ImagePool, and the StringBuffer pool SBPool. When we create a pool, we will add it to the pool list by calling the PoolList.add() method, and when we want to retrieve a pool, we will call PoolList.getPool() and use the name of the pool as a unique identifier. Listing 7.9 shows the code for PoolList.

**Listing 7.9  The PoolList class**

```java
package enterprise.common;
import java.util.*;

public class PoolList
{
    private Hashtable m_poolList = new Hashtable();

    public synchronized void add(String name, Pool pool)
        throws IllegalArgumentException
    {
        if (pool == null)
        {
            throw new IllegalArgumentException("Pool object is null.");
        }

        if (name == null)
        {
            throw new IllegalArgumentException("name is null.");
        }

        if (m_poolList.containsKey(name))
        {
            throw new IllegalArgumentException(name + " already exists.");
        }

        m_poolList.put(name, pool);
    }

    public synchronized Pool getPool(String name)
        throws IllegalArgumentException
    {
        if (name == null)
        {
            throw new IllegalArgumentException("name is null.");
        }

        if (m_poolList.containsKey(name))
        {
            throw new IllegalArgumentException(name + " already exists.");
        }

        return m_poolList.get(name);
    }
}
```
Now we need the equivalent of a customer service representative—a pool manager—to manage this pool list, monitoring which objects are checked out from which pool and ensuring that objects are checked back in. We’ll call the object we create for this purpose PoolObjectManager. It will field all requests to check out and check in objects from a pool, rather than the pool fielding such requests directly. PoolObjectManager will keep track of all the objects that have been checked out by storing them in a list. As the user requires another object, PoolObjectManager will check out the object and add it to its internal array. If the user manually checks in an object, PoolObjectManager will remove the object from the internal list.

When the developer is done executing the call, the checkInAll() method is executed in PoolObjectManager, which goes through the internal object list and checks in any objects that have not already been checked in. The object PoolObjectManager ensures that the developer never directly accesses a pool. Instead, the developer requests a pool by a particular name (such as SBPool) from PoolObjectManager and gets an object back. When the code is finished executing, PoolObjectManager checks in all checked-out objects. Listing 7.10 shows the code for PoolObjectManager.

Listing 7.10  The PoolObjectManager class

```java
package enterprise.common;
import java.util.*;
public class PoolObjectManager
{
    private Hashtable m_checkedOut = new Hashtable();
    private PoolList m_poolList = null;
    public PoolObjectManager(PoolList pl)
    {   m_poolList = pl;
    }
    public Object checkOut(String name) throws Exception
    {
        if (m_poolList.containsKey(name) == false)
        {
            throw new IllegalArgumentException(name +
                " does not exist in list.");
        }
        return (Pool) m_poolList.get(name);
    }
}
```
Pool p = m_poolList.getPool(name);
Object poolObject = p.checkOutPoolObject();
if (poolObject != null)
{
    m_checkedOut.put(poolObject, p);
}
return poolObject;
}
public void checkIn(Object o)
{
    Pool p = (Pool)m_checkedOut.get(o);
    p.checkInPoolObject(o);
    m_checkedOut.remove(o);
}
public void checkInAll()
{
    Enumeration e = m_checkedOut.keys();
    while (e.hasMoreElements())
    {
        Object o = e.nextElement();
        checkIn(o);
    }
}

Now that we have created PoolList and PoolObjectManager, what do we do with them? How do we use pools with them? The only way we can use these objects is to locate them centrally. PoolList is specific to an application. Each application has its own list of pools, just as a video store has its own list of videos. This makes PoolList a good candidate to be a member of AppContext. So let's create a PoolList member in AppContext, as well as a getter for the object. These additions to AppContext are shown in Listing 7.11.

Listing 7.11 Changes to AppContext to provide access to PoolList
public class AppContext
{
    .
    .
    .
    private PoolList oPoolList = new PoolList();
    .
    .

public PoolList getPoolList()
{
    return oPoolList;
}

PoolObjectManager is where we go when we want to check out an object; therefore we must have access to this object from our dispatch methods. For such access to be possible, the HTTPMethod object needs a member variable of type PoolObjectManager. This member variable, which we will call m_poolObjMgr, will be set by BaseEnterpriseServlet when it calls HTTPMethod.registerMethod(). We will create the member variable and alter the registerMethod() method to accept PoolObjectManager as one of its parameters, and set the member variable from within this method. Listing 7.12 shows these changes.

Listing 7.12  Changes to the HTTPMethod class to provide access to the PoolObjectManager object

    public abstract class HTTPMethod implements Authentication
    {
        
        protected PoolObjectManager m_poolObjMgr;
        
        public void registerMethod(HttpServletRequest  req,
                                      HttpServletResponse res,
                                      AppContext context,
                                      PoolObjectManager pom)
        {
        
        m_poolObjMgr = pom;
        
        }
        
        Now we’ll see the real magic happen. The main engine for PoolList and PoolObjectManager is in BaseEnterpriseServlet. We will take a little closer look at BaseEnterpriseServlet to see what changes need to be made so that
PoolList and PoolObjectManager can be used. Because the dispatching occurs in the service() method of BaseEnterpriseServlet, we will start there.

```java
public abstract class BaseEnterpriseServlet extends HttpServlet {
    ...
    public void service(HttpServletRequest req,
                      HttpServletResponse res)
        throws ServletException, IOException {
    ...

    We know that all pools require a try/finally clause of some sort. There is no exception here. In this method there is already a large try/catch block, so we will reuse this block with the pool code. Right before the try clause we want to create a PoolObjectManager object so that any checked-out objects can be monitored.

    We create PoolObjectManager and use PoolList to construct the object. Understand that PoolObjectManager needs to know about all available pools, which is why we pass PoolList to it. By doing this, the developers can ask PoolObjectManager if it can check out SBPool, ImagePool, or any other specific pool that may have been created in the application. Because PoolList is accessible through AppContext, we would create PoolObjectManager with the following code:

```java
    PoolObjectManager pom =
        new PoolObjectManager(m_ac.getPoolList());
    try {
    ...
```

The rest of the try block and code are the same, until we get to registerMethod(). In the try clause, we will call registerMethod() with PoolObjectManager added as one of the parameters because we changed HTTPMethod's version to accept this parameter. This gives HTTPMethod access to PoolObjectManager.

```java
    methodInstance.registerMethod(req, res, m_ac, pom);
    ...
```
At the end of the catch blocks we want to add a finally block because we are allowing the dispatched methods to check out any object they want from PoolObjectManager, and PoolObjectManager must clean up afterward, even if an exception is thrown in the dispatch method. The finally clause takes PoolObjectManager and calls the checkInAll() method, which checks in any objects that were checked out by the dispatch method.

```java
} finally
{
    // Close up the connections
    pom.checkInAll();
}
```

For monitoring checkout of objects, we may want to give access to PoolObjectManager in the initializeApp() and destroyApp() methods of BaseEnterpriseServlet. When we extend BaseEnterpriseServlet, we may want the initialization and destruction code to have access to any pools. Although we will create most pools in the initializeApp() method, when we get into databases and pooling, we may offload some of this work to AppContext, so initializeApp() may need access to some pools that have already been created. For this reason we will allow the initializeApp() and destroyApp() methods to take PoolObjectManager as a parameter. Of course, the code that calls these methods will need the usual try/finally block, and it will need to create and pass PoolObjectManager to the initializeApp() and destroyApp() methods. The changes are as follows:

```java
public abstract void initializeApp(AppContext appContext,
        PoolObjectManager pom)
        throws Exception;

public abstract void destroyApp(AppContext appContext,
        PoolObjectManager pom);
```

```java
public final void init(ServletConfig config) throws ServletException
{
    ...
    m_ac.parseConfigFile(getAppName(), sConf);
    PoolObjectManager pom = new PoolObjectManager(m_ac.getPoolList());
```
try {
    initializeApp(m_ac, pom);
} finally {
    pom.checkInAll();
}

.
.
.
public final void destroy() {
    if (getAppName() != null) {
        m_appManager.removeApp(getAppName());
    }

    PoolObjectManager pom = new PoolObjectManager(m_ac.getPoolList());

    try {
        destroyApp(m_ac, pom);
    } finally {
        pom.checkInAll();
    }

What does all this code do for us? It takes the burden of having to check in pool objects away from the developer and puts it in the hands of BaseEnterpriseServlet. To use a pool with this framework, we would create the pool and add it to the list with the following code:

```
SBOPool sboPool = new SBOPool(5);
appContext.getPoolList().add("SBOPPOOL", sboPool);
```

In the dispatch class, we check out an object like this:

```
SBOObject sbo = (SBOobject) m_poolObjMgr.checkOut("SBOPPOOL");
```
That's all there is to it! No need for any try/finally blocks, and more importantly, no need to remember to check the object back into the pool. BaseEnterpriseServlet and PoolObjectManager do all the dirty work.

Using PoolList and PoolObjectManager: An Example

Creating and using pools with PoolList and PoolObjectManager is really easy, but don't take my word for it. Let's take the PoolExample application that we developed earlier in the chapter and change it so that it uses PoolList and PoolObjectManager. We'll start with the PoolExample base dispatch servlet. We no longer need the SBOPool static member in the class. Remember that we now store the pools in the application's PoolList object. We still initialize the pool in the initializeApp() method, but we also add the pool to PoolList through AppContext. We need to give the pool a string name when we add it to the list, so we will call it SBOPool. Listing 7.13 shows the code for the new PoolExample class.

Listing 7.13  The new PoolExample class

```java
import enterprise.servlet.*;
import enterprise.common.*;
import java.io.*;

public class PoolExample extends BaseEnterpriseServlet
{
    public String getAppName()
    {
        return "PoolExample";
    }

    public void initializeApp(AppContext appContext, PoolObjectManager pom) throws Exception
    {
        SBOPool sboPool = new SBOPool(5);
        appContext.getPoolList().add("SBOPool", sboPool);
    }

    public void destroyApp(AppContext ac, PoolObjectManager pom)
    {
    }
}
```

The only other code we need to change is in the PoolPage class. We can strip out the try/finally block and request the object SBObject from the PoolObjectManager member variable m_poolObjMgr. We request the pool by the name that we gave it when we created it. We call checkout() with this string...
as a parameter, and it gives us the object we requested. Listing 7.14 shows the code for the new PoolPage class.

**Listing 7.14** The new PoolPage class

```java
import enterprise.servlet. *
import enterprise.html. *
import java.io. *

public class PoolPage extends HTTPMethod {

    public void execute() throws Exception {
        PrintWriter out = m_response.getWriter();
        TemplateCacheList tcl = m_context.getTemplateCacheList();
        HTMLCompiledTemplate hct = tcl.get("pool");
        HTMLTemplate ht = hct.createHTMLTemplate();
        SBObject sbo = (SBObject) m_poolObjMgr.checkOut("SBOPOOL");
        ht.substitute("poolno", sbo.bufferId + ";
        ht.substitute("BASEURL", m_context.getAppBaseURL());
        ht.toStringBuffer(sbo.sb);
        out.println(sbo.sb.toString());
    }
}
```

Notice that the revised code for PoolPage does not check in the SBObject object. PoolObjectManager takes care of this. We could check in the object by calling the PoolObjectManager.checkIn() method, but this is not necessary because the object will clean up after us. Although nothing prevents us from creating or accessing pools directly, using pools in conjunction with PoolList and PoolObjectManager ensures that the developer does not forget to check in objects. In addition, as this example shows, a lot less code needs to be implemented in the dispatch classes, so this approach speeds up development time.

**Summary**

In this chapter we learned about object pools and developed the base pool object that allows us to create a pool with just about any kind of object. We also learned that creating a pool places responsibility on the developer to make sure that each piece of code that uses a pooled object checks used objects back in. If objects are not checked back in, the pool becomes depleted of available
resources, and the Web server eventually gives the impression of freezing up as the threads wait for available objects.

We created PoolList and PoolObjectManager to help alleviate this burden. When using pools with these objects in the enterprise servlet framework, we no longer need to worry about checking in pooled objects when we’re done using them. PoolObjectManager monitors and checks in the checked-out objects after we’re done using them.

As this chapter showed, object creation and destruction can sometimes be considered the enemy in Java programming. Large blocks of memory resources or lengthy connection processes can hinder a Web server’s performance. The garbage collector can also slow things down. By pooling resources and reusing objects, we can streamline applications and provide scalability to servlets.