Lazy instantiation

Balancing performance and resource usage

Summary
The balance between the performance of software and the resources it consumes has been an issue since the birth of computers. It stems from the need to conserve valuable resources such as memory -- possibly at the expense of performance. This tip examines one way to reduce memory usage in Java programs by deferring the creation of objects until they are actually required. *(2,400 words)*

By Philip Bishop and Nigel Warren

It wasn't so long ago that we were thrilled by the prospect of having the on-board memory in an 8-bit microcomputer jump from 8 KB to 64 KB. Judging by the ever increasing, resource-hungry applications we now use, it's amazing that anybody ever managed to write a program to fit into that tiny amount of memory. While we have much more memory to play with these days, some valuable lessons can be learned from the techniques established to work within such tight constraints.

Moreover, Java programming isn't just about writing applets and applications for deployment on personal computers and workstations; Java has made strong inroads into the embedded systems market as well. Current embedded systems have relatively scarce memory resources and computing power, so many of the old issues facing programmers have resurfaced for Java developers working in the device realm.

Balancing these factors is a fascinating design problem: it's important to accept the fact that no solution in the area of embedded design will be perfect. So, we need to understand the types of techniques that are going to be useful in achieving the fine balance required to work within the constraints of the deployment platform.

One of the memory conservation techniques Java programmers find useful is *lazy instantiation.* With lazy instantiation, a program refrains from creating certain resources until the resource is first needed -- freeing valuable memory space. In this tip, we examine lazy instantiation techniques in Java class loading and object creation, and the special considerations required for Singleton patterns. The material in this tip derives from the work in Chapter 9 of our book, *Java in Practice: Design Styles & Idioms for Effective Java* (see Resources).

Eager vs. lazy instantiation: an example

If you're familiar with Netscape's Web browser and have used both versions 3.x and 4.x, undoubtedly you've noticed a difference in how the Java runtime is loaded. If you look at the splash screen when Netscape 3 starts up, you'll note that it loads various resources, including Java. However, when you start up Netscape 4.x, it doesn't load the Java runtime -- it waits until you visit a Web page that includes the <APPLET> tag. These two approaches illustrate the techniques of *eager instantiation* (load it in case it's needed) and *lazy instantiation* (wait until it's requested before you load it, as it may never be needed).

There are drawbacks to both approaches: on one hand, always loading a resource potentially wastes precious memory if the resource isn't used during that session; on the other hand, if it hasn't been loaded, you pay the price in terms of loading time when the resource is first required.

Consider lazy instantiation as a resource conservation policy

Lazy instantiation in Java falls into two categories:

- Lazy class loading
- Lazy object creation

Lazy class loading

The Java runtime has built-in lazy instantiation for classes. Classes load into memory only when they're first referenced. (They also may be loaded from a Web server via HTTP first.)

```
MyUtils.classMethod();    //first call to a static class method
Vector v = new Vector(); //first call to operator new
```

Lazy class loading is an important feature of the Java runtime environment as it can reduce memory usage under certain circumstances. For example, if a part of a program never is executed during a session, classes referenced only in that part of the program never will be loaded.

Lazy object creation

Lazy object creation is tightly coupled to lazy class loading. The first time you use the new keyword on a class type that previously hasn't been loaded, the Java runtime will load it for you. Lazy object creation can reduce memory usage to a much greater extent than lazy class loading.

To introduce the concept of lazy object creation, let's take a look at a simple code example where a `Frame` uses a `MessageBox` to display error messages:

```
public class MyFrame extends Frame
{
  private MessageBox mb_ = new MessageBox();

  //private helper used by this class
  private void showMessage(String message)
  {
    //set the message text
    mb_.setMessage( message );
    mb_.pack();
    mb_.show();
  }
}
```

In the above example, when an instance of `MyFrame` is created, the `MessageBox` instance mb_ is also created. The same rules apply recursively. So any instance variables initialized or assigned in class

MessageBox's constructor also are allocated off the heap and so on. If the instance of MyFrame isn't used to display an error message within a session, we're wasting memory unnecessarily.

In this rather simple example, we aren't really going to gain too much. But if you consider a more complex class, which uses many other classes, which in turn use and instantiate more objects recursively, the potential memory usage is more apparent.

Consider lazy instantiation as a policy to reduce resource requirements

The lazy approach to the above example is listed below, where the `object mb_` is instantiated on the first call to `showMessage()`. (That is, not until it's actually needed by the program.)

```
public final class MyFrame extends Frame
{
  private MessageBox mb_ ; //null, implicit

  //private helper used by this class
  private void showMessage(String message)
  {
    if(mb_==null)//first call to this method
      mb_=new MessageBox();

    //set the message text
    mb_.setMessage( message );
    mb_.pack();
    mb_.show();
  }
}
```

If you take a closer look at `showMessage()`, you'll see that we first determine whether the instance variable mb_ is equal to null. As we haven't initialized mb_ at its point of declaration, the Java runtime has taken care of this for us. Thus, we can safely proceed by creating the `MessageBox` instance. All future calls to `showMessage()` will find that mb_ is not equal to null, therefore skipping the creation of the object and using the existing instance.

A real-world example

Let's now examine a more realistic example, where lazy instantiation can play a key role in reducing the amount of resources used by a program.

Assume that we have been asked by a client to write a system that will let users catalog images on a filesystem and provide the facility to view either thumbnails or complete images. Our first attempt might be to write a class that loads the image in its constructor.

```
public class ImageFile
{
  private String filename_;
  private Image image_;

  public ImageFile(String filename)
  {
    filename_=filename;
    //load the image
  }
  public String getName(){ return filename_;}
  public Image getImage()
```

```
  {
    return image_;
  }
}
```

In the example above, `ImageFile` implements an overeager approach to instantiating the `Image` object. In its favor, this design guarantees that an image will be available immediately at the time of a call to `getImage()`. However, not only could this be painfully slow (in the case of a directory containing many images), but this design could exhaust the available memory. To avoid these potential problems, we can trade the performance benefits of instantaneous access for reduced memory usage. As you may have guessed, we can achieve this by using lazy instantiation.

Here's the updated `ImageFile` class using the same approach as class `MyFrame` did with its `MessageBox` instance variable:

```
public class ImageFile
{
  private String filename_;
  private Image image_; //=null, implicit

  public ImageFile(String filename)
  {
    //only store the filename
    filename_=filename;
  }
  public String getName(){ return filename_;}
  public Image getImage()
  {
    if(image_==null)
    {
      //first call to getImage()
      //load the image...
    }
    return image_;
  }
}
```

In this version, the actual image is loaded only on the first call to `getImage()`. So to recap, the trade-off here is that to reduce the overall memory usage and startup times, we pay the price for loading the image the first time it is requested -- introducing a performance hit at that point in the program's execution. This is another idiom that reflects the `Proxy` pattern in a context that requires a constrained use of memory.

The policy of lazy instantiation illustrated above is fine for our examples, but later on you'll see how the design has to alter in the context of multiple threads.

Lazy instantiation for Singleton patterns in Java

Let's now take a look at the Singleton pattern. Here's the generic form in Java:

```
public class Singleton
{
  private Singleton() {}

  static private Singleton instance_
    = new Singleton();
```

```
  static public Singleton instance()
  {
    return instance_;
  }
  //public methods
}
```

In the generic version, we declared and initialized the `instance_` field as follows:

```
static final Singleton instance_ = new Singleton();
```

Readers familiar with the C++ implementation of Singleton written by the GoF (the Gang of Four who wrote the book *Design Patterns: Elements of Reusable Object-Oriented Software* -- Gamma, Helm, Johnson, and Vlissides) may be surprised that we didn't defer the initialization of the `instance_` field until the call to the `instance()` method. Thus, using lazy instantiation:

```
public Singleton instance()
{
  if(instance_==null) //Lazy instantiation
    instance_= new Singleton();

  return instance_;
}
```

The listing above is a direct port of the C++ Singleton example given by the GoF, and frequently is touted as the generic Java version too. If you already are familiar with this form and were surprised that we didn't list our generic Singleton like this, you'll be even more surprised to learn that it is totally unnecessary in Java! This is a common example of what can occur if you port code from one language to another without considering the respective runtime environments.

For the record, the GoF's C++ version of Singleton uses lazy instantiation because there is no guarantee of the order of static initialization of objects at runtime. (See Scott Meyer's Singleton for an alternative approach in C++ .) In Java, we don't have to worry about these issues.

The lazy approach to instantiating a Singleton is unnecessary in Java because of the way in which the Java runtime handles class loading and static instance variable initialization. Previously, we have described how and when classes get loaded. A class with only public static methods gets loaded by the Java runtime on the first call to one of these methods; which in the case of our Singleton is

```
Singleton s=Singleton.instance();
```

The first call to `Singleton.instance()` in a program forces the Java runtime to load the class `Singleton`. As the field `instance_` is declared as static, the Java runtime will initialize it after successfully loading the class. Thus guarantees that the call to `Singleton.instance()` will return a fully initialized Singleton -- get the picture?

Lazy instantiation: dangerous in multithreaded applications

Using lazy instantiation for a concrete Singleton is not only unnecessary in Java, it's downright dangerous in the context of multithreaded applications. Consider the lazy version of the `Singleton.instance()` method, where two or more separate threads are attempting to obtain a reference to the object via `instance()`. If one thread is preempted after successfully executing the line `if(instance_==null)`, but before it has completed the line `instance_=new Singleton()`, another thread can also enter this method with `instance_ still ==null` -- nasty!

The outcome of this scenario is the likelihood that one or more Singleton objects will be created. This is a major headache when your Singleton class is, say, connecting to a database or remote server. The simple solution to this problem would be to use the synchronized key word to protect the method from multiple threads entering it at the same time:

```
synchronized static public instance() {...}
```

However, this approach is a bit heavy-handed for most multithreaded applications using a Singleton class extensively, thereby causing blocking on concurrent calls to `instance()`. By the way, invoking a synchronized method is always much slower than invoking a nonsynchronized one. So what we need is a strategy for synchronization that doesn't cause unnecessary blocking. Fortunately, such a strategy exists. It is known as the *double-check idiom.*

The double-check idiom

Use the double-check idiom to protect methods using lazy instantiation. Here's how to implement it in Java:

```
public Singleton instance()
{
  if(instance_==null) //don't want to block here
  {
    //two or more threads might be here!!!

    synchronized(Singleton.class)
    {
      //must check again as one of the
      //blocked threads can still enter

      if(instance_==null)
        instance_= new Singleton();//safe
    }
  }
  return instance_;
}
```

The double-check idiom improves performance by using synchronization only if multiple threads call `instance()` before the Singleton is constructed. Once the object has been instantiated, `instance_` is no longer `==null`, allowing the method to avoid blocking concurrent callers.

Using multiple threads in Java can be very complex. In fact, the topic of concurrency is so vast that Doug Lea has written a whole book on it: *Concurrent Programming in Java.* If you're new to concurrent programming, we recommend you get a copy of this book before you embark on writing complex Java systems that rely on multiple threads.

Conclusion

Frequently, trade-offs must be made to satisfy users expectations or to utilize scarce resources efficiently. Understanding the technique for implementing lazy instantiation in Java, coupled with the rich set of features offered by the Java language and runtime environment, can help you design elegant, efficient, and fast software systems. Eager instantiation along with lazy and eager evaluation are investigated in depth in our book. ■

Also this month in JavaWorld
Go

Resources

- Read *Java in Practice: Design Styles & Idioms for Effective Java* by Nigel Warren and Philip Bishop (Addison Wesley Longman, ISBN: 0201360659) http://www1.clbooks.com/asp/bookinfo/bookinfo.asp?theisbn=0201360659
- Be sure to check out *Concurrent programming in Java: Design Principles and Patterns* by Doug Lea (Addison Wesley Longman, ISBN: 0201695812) http://www1.clbooks.com/asp/bookinfo/bookinfo.asp?theisbn=0201695812
- And don't miss *More effective C++* by Scott Meyers (Addison Wesley Longman, ISBN: 020163371X) http://www1.clbooks.com/asp/bookinfo/bookinfo.asp?theisbn=020163371X

About the author
Philip Bishop is an independent distributed object consultant specializing in the design and implementation of Java and CORBA systems. philip.bishop@javaworld.com

Nigel Warren is chief technical architect at Digital Bridges Ltd., where he works with Java to build distributed service platforms. nigel.warren@javaworld.com.

Both authors have consulted many international companies on object oriented systems design and joined forces to write *Java in Practice -- Design Styles & Idioms for Effective Java* published by Addison Wesley.

What did you think of this article?

-Very worth reading	-Too long	-Too technical
-Worth reading	-Just right	-Just right
-Not worth reading	-Too short	-Not technical enough

Comments:
Name:
Email:
Company Name: Send data

Last modified: Friday, January 22, 1999