```
    }

    @Override
    protected Class findClass(String name) throws ClassNotFoundException
    {
        throw new ClassNotFoundException(name);
    }
}
```

The `ClassLoader` in Example 9-3 holds no references to the classes it defines. Each class is a singleton with no relations, and so `findClass` is safe to throw `ClassNotFoundException` and all is right with the world. Once the singleton instance of each class becomes a GC candidate, so will the class itself, and both can be collected. This works great, and the classes all get loaded. However, something goes wrong. For some unknown reason, no classes get unloaded, ever. It appears that some object somewhere is holding onto references to our classes.

Let's look down the list of calls on defining a new class:

```
java.lang.ClassLoader: defineClass(...)
java.lang.ClassLoader: defineClass1(...)
ClassLoader.c: Java_java_lang_ClassLoader_defineClass1(...)
vm/prims/jvm.cpp: JVM_DefineClassWithSource(...)
vm/prims/jvm.cpp: jvm_define_class_common(...)
vm/memory/systemDictionary.cpp: SystemDictionary::resolve_from_stream(...)
vm/memory/systemDictionary.cpp: SystemDictionary::define_instance_class(...)
```

This is what happens when you're not satisfied with the answer "because it does." We get down this far into the guts of the JVM, and we find this little snippet of code:

```
// Register class just loaded with class loader (placed in Vector)
// Note we do this before updating the dictionary, as this can
// fail with an OutOfMemoryError (if it does, we will *not* put this
// class in the dictionary and will not update the class hierarchy).
if (k->class_loader() != NULL) {
  methodHandle m(THREAD, Universe::loader_addClass_method());
  JavaValue result(T_VOID);
  JavaCallArguments args(class_loader_h);
  args.push_oop(Handle(THREAD, k->java_mirror()));
  JavaCalls::call(&result, m, &args, CHECK);
}
```

which is a call back up to the Java level into the classloader instance that is doing the loading:

```
// The classes loaded by this class loader.  The only purpose of this table
// is to keep the classes from being GC'ed until the loader is GC'ed.
private Vector classes = new Vector();

// Invoked by the VM to record every loaded class with this loader.
void addClass(Class c) {
    classes.addElement(c);
}
```

So now we know which naughty little object is holding a reference to all of our classes and keeping them from being garbage collected. Unfortunately, there is little we can do about this.