Item 3:

**Enforce the singleton property with a private**

**constructor or an enum type**

a privileged client can invoke the private constructor reflectively

(Item 53) with the aid of the AccessibleObject.setAccessible method.

If you

need to defend against this attack, modify the constructor to make it throw an

exception if it’s asked to create a second instance.

To make a singleton class that is implemented using either of the previous

approaches *serializable* (Chapter 11), it is not sufficient merely to add implements

Serializable to its declaration. To maintain the singleton guarantee, you

have to declare all instance fields transient and provide a readResolve method

(Item 77). Otherwise, each time a serialized instance is deserialized, a new

instance will be created, leading, in the case of our example, to spurious Elvis

sightings. To prevent this, add this readResolve method to the Elvis class:

**// readResolve method to preserve singleton property**

private Object readResolve() {

// Return the one true Elvis and let the garbage collector

// take care of the Elvis impersonator.

return INSTANCE;

}

**a single-element enum type is the best way to implement a singleton**.

**// Enum singleton - the preferred approach**

public enum Elvis {

INSTANCE;

public void leaveTheBuilding() { ... }

}

**Item 4: Enforce noninstantiability with a private constructor**

**// Noninstantiable utility class**

public class UtilityClass {

**// Suppress default constructor for noninstantiability**

private UtilityClass() {

throw new AssertionError();

}

... // Remainder omitted

}

The AssertionError isn’t strictly required, but it provides insurance in

case the constructor is accidentally invoked from within the class. It guarantees

that the class will never be instantiated under any circumstances. This idiom is

mildly counterintuitive, as the constructor is provided expressly so that it cannot

be invoked. It is therefore wise to include a comment, as shown above.

**Item 5: Avoid creating unnecessary objects**

An object can always be reused if it is *immutable* (Item 15).

String s = new String("stringette"); **// DON'T DO THIS!**

String s = "stringette";

This version uses a single String instance, rather than creating a new one

each time it is executed. Furthermore, it is guaranteed that the object will be

reused by any other code running in the same virtual machine that happens to contain

the same string literal [JLS, 3.10.5].

You can often avoid creating unnecessary objects by using *static factory methods*

(Item 1) in preference to constructors on immutable classes that provide both.

For example, the static factory method Boolean.valueOf(String) is almost

always preferable to the constructor Boolean(String). The constructor creates a

new object each time it’s called, while the static factory method is never required

to do so and won’t in practice.

**// Hideously slow program! Can you spot the object creation?**

public static void main(String[] args) {

Long sum = 0L;

for (long i = 0; i < Integer.MAX\_VALUE; i++) {

sum += i;

}

System.out.println(sum);

}

This program gets the right answer, but it is *much* slower than it should be,

due to a one-character typographical error. The variable sum is declared as a Long

instead of a long, which means that the program constructs about 231 unnecessary

Long instances (roughly one for each time the long i is added to the Long sum).

Changing the declaration of sum from Long to long reduces the runtime from 43

seconds to 6.8 seconds on my machine. The lesson is clear: **prefer primitives to**

**boxed primitives, and watch out for unintentional autoboxing.**

**Item 6: Eliminate obsolete object references**

**// Can you spot the "memory leak"?**

public class Stack {

private Object[] elements;

private int size = 0;

private static final int DEFAULT\_INITIAL\_CAPACITY = 16;

public Stack() {

elements = new Object[DEFAULT\_INITIAL\_CAPACITY];

}

public void push(Object e) {

ensureCapacity();

elements[size++] = e;

}

public Object pop() {

if (size == 0)

throw new EmptyStackException();

return elements[--size];

}

/\*\*

\* Ensure space for at least one more element, roughly

\* doubling the capacity each time the array needs to grow.

\*/

private void ensureCapacity() {

if (elements.length == size)

elements = Arrays.copyOf(elements, 2 \* size + 1);

}

}

the objects

that were popped off the stack will not be garbage collected, even if the program

using the stack has no more references to them. This is because the stack maintains

*obsolete references* to these objects. An obsolete reference is simply a reference

that will never be dereferenced again. In this case, any references outside of

the “active portion” of the element array are obsolete.

Memory leaks in garbage-collected languages (more properly known as *unintentional*

*object retentions*) are insidious. If an object reference is unintentionally

retained, not only is that object excluded from garbage collection, but so too are

any objects referenced by that object, and so on.

The fix for this sort of problem is simple: null out references once they

become obsolete. In the case of our Stack class, the reference to an item becomes

obsolete as soon as it’s popped off the stack. The corrected version of the pop

method looks like this:

public Object pop() {

if (size == 0)

throw new EmptyStackException();

Object result = elements[--size];

**elements[size] = null; // Eliminate obsolete reference**

return result;

}

**Nulling out object references should be the exception rather than the norm.**

The best way to eliminate an obsolete reference is to let the variable that contained

the reference fall out of scope. This occurs naturally if you define each variable in

the narrowest possible scope (Item 45).

Generally speaking, **whenever a class manages its own memory, the programmer**

**should be alert for memory leaks**. Whenever an element is freed, any

object references contained in the element should be nulled out.

**Item 7: Avoid finalizers**

One shortcoming of finalizers is that there is no guarantee they’ll be executed

promptly [JLS, 12.6]. It can take arbitrarily long between the time that an object

becomes unreachable and the time that its finalizer is executed. This means that

you should **never do anything time-critical in a finalizer.**