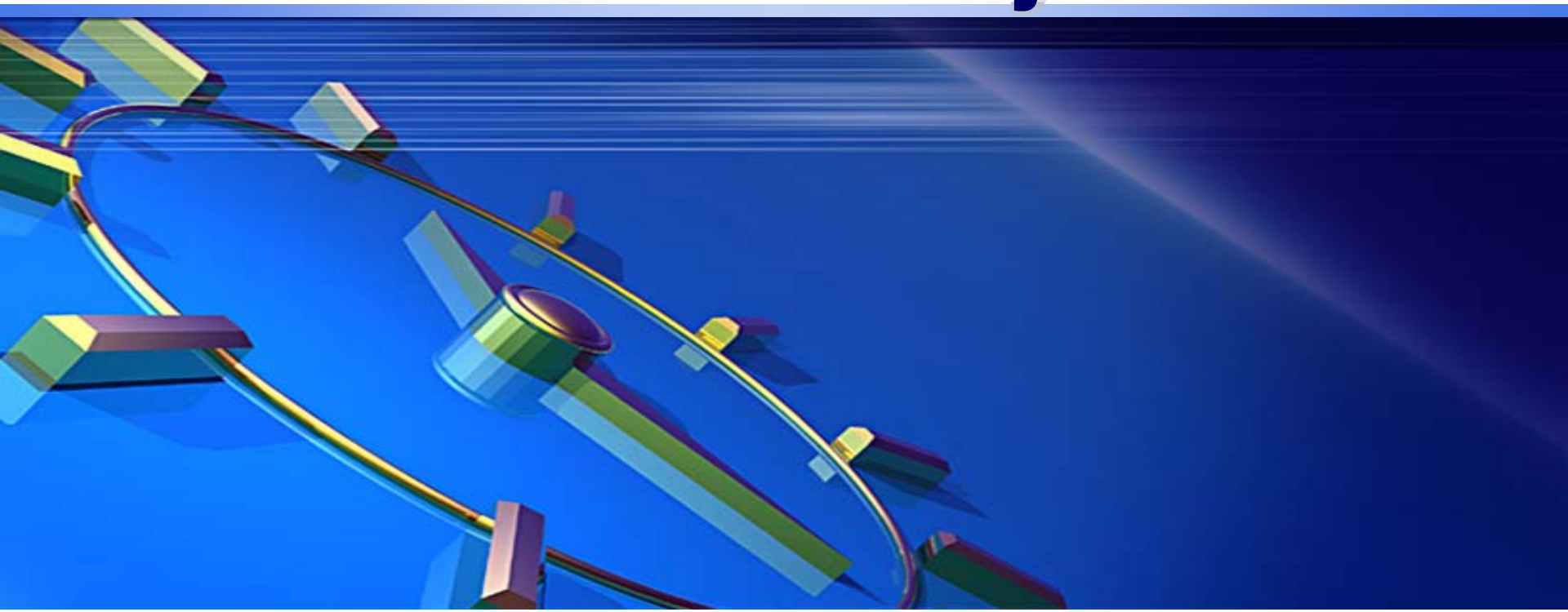
LOGO

Finalization and Reference Objects



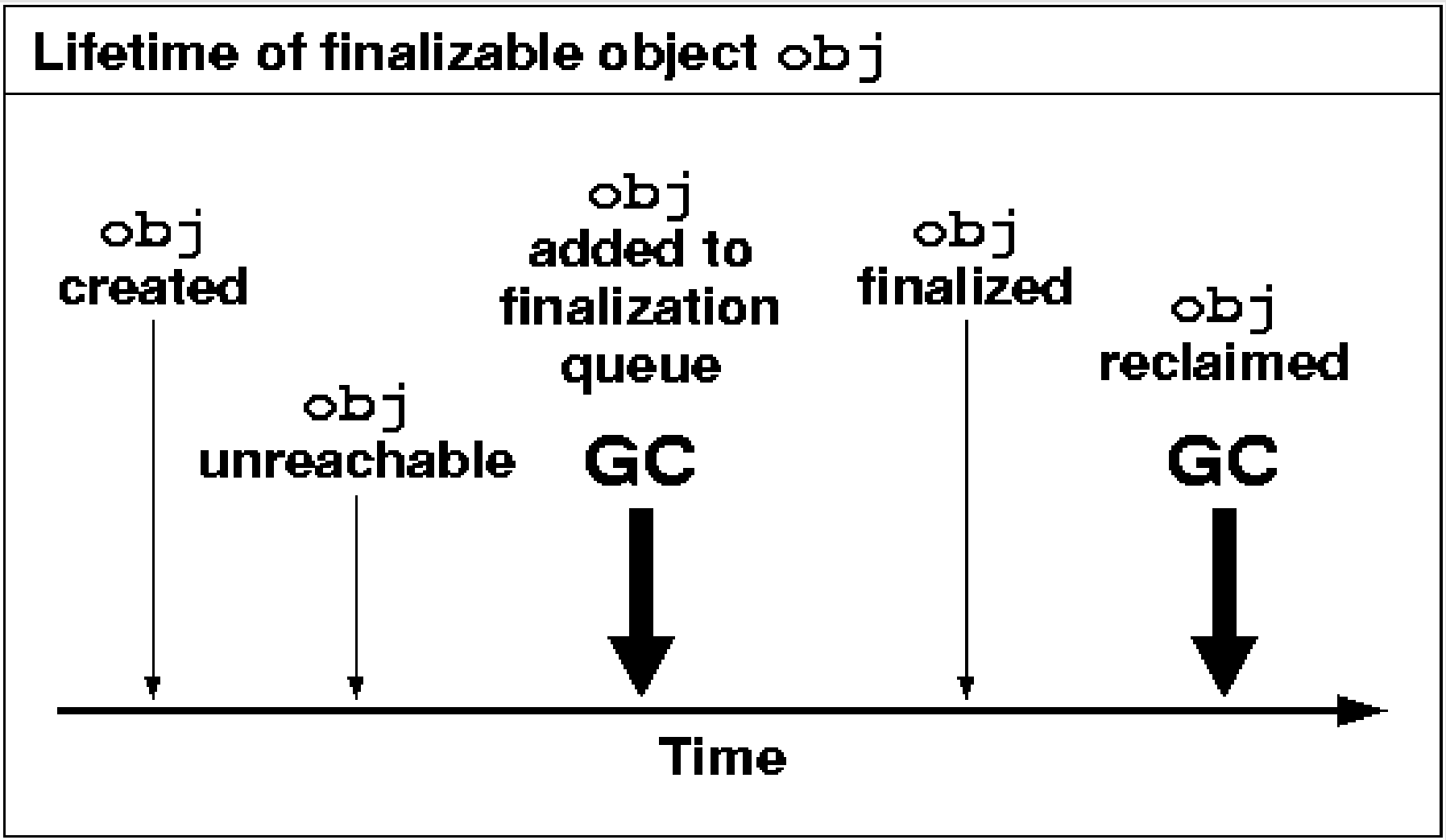
By Mohit Kumar

Finalization

- Finalization is a feature of the Java programming language that allows you to perform postmortem cleanup on objects that the garbage collector has found to be unreachable. It is typically used to reclaim native resources associated with an object.

```
public class Image1 {
          // pointer to the native image data
          private int nativeImg;
          private Point pos;
          private Dimension dim;

          // it disposes of the native image;
          // successive calls to it will be ignored
          private native void disposeNative();
          public void dispose() { disposeNative(); }
          protected void finalize() { dispose(); }

          static private Image1 randomImg;
}
```

Lifetime of finalizable objects



Finalization Process

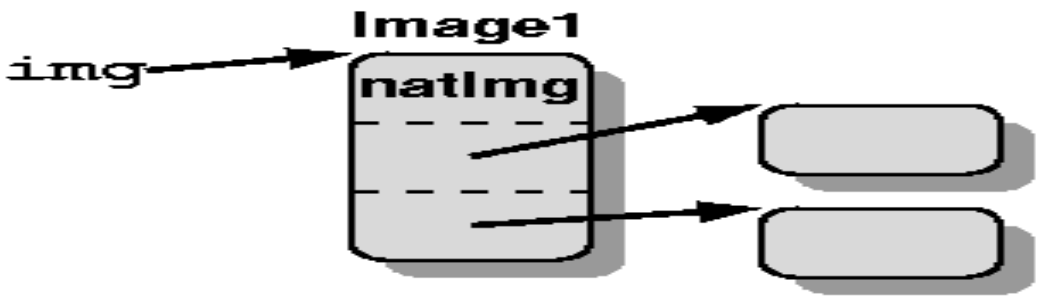
- When obj is allocated, the JVM internally records that obj is finalizable. This typically slows down the otherwise fast allocation path that modern JVMs have.
- When the garbage collector determines that obj is unreachable, it notices that obj is finalizable -- as it had been recorded upon allocation -- and adds it to the JVM's *finalization queue*. It also ensures that all objects reachable from obj are retained, even if they are otherwise unreachable, as they might be accessed by the finalizer.

Finalization Process

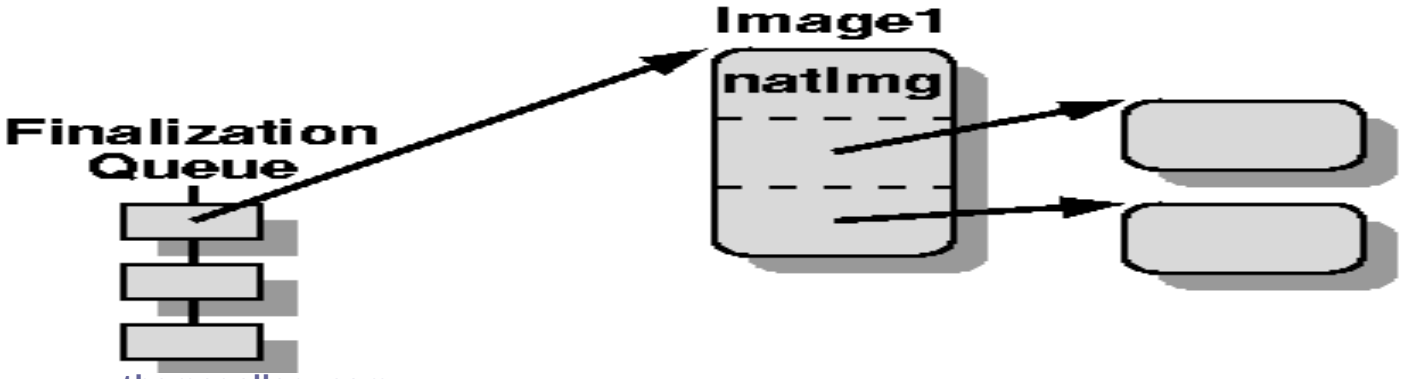
- At some point later, the JVM's *finalizer thread* will dequeue obj, call its finalize() method, and record that the obj's finalizer has been called. At this point, obj is considered to be *finalized*.
- When the garbage collector rediscovers that obj is unreachable, it will reclaim its space along with everything reachable from it, provided that the latter is otherwise unreachable

Finalization Process

```
img = new Image1();
```




```
img = null;
```
and after a subsequent GC...



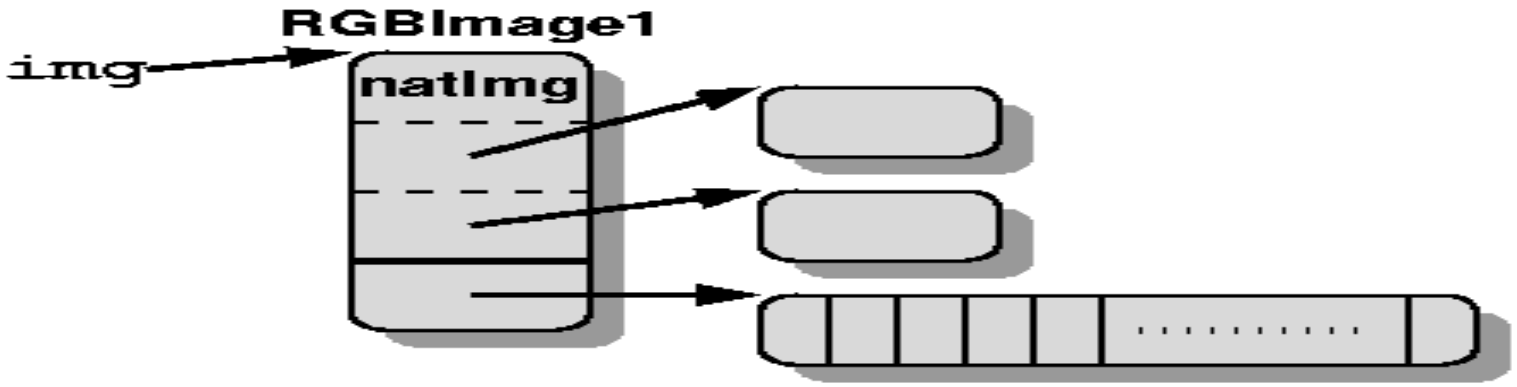
Finalization Details and Pitfalls

- Finalization can delay the reclamation of resources, even if you do not use it explicitly.

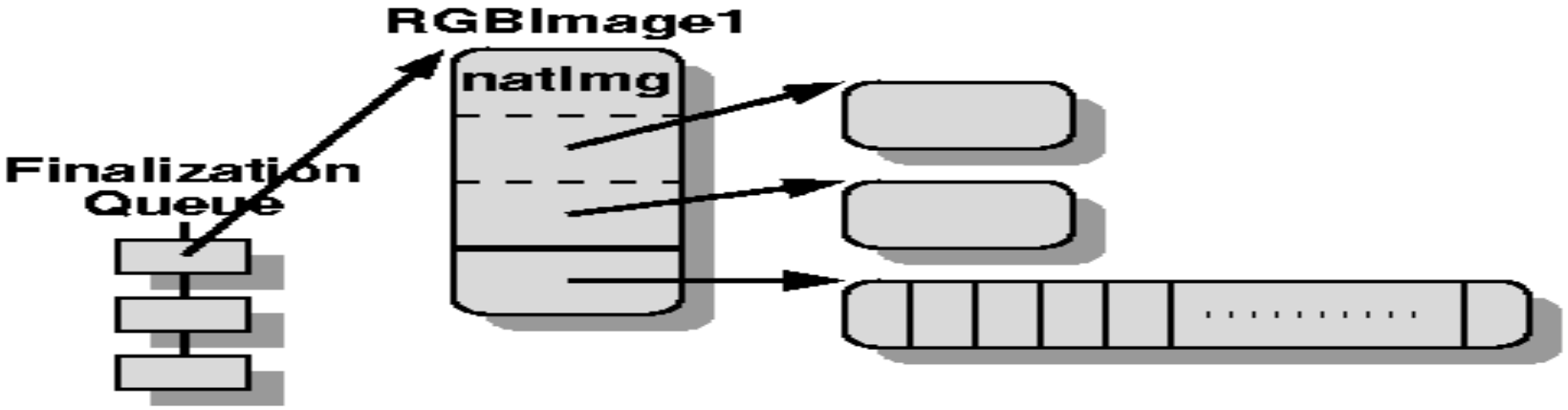
```
public class RGBImage1 extends Image1 {
        private byte rgbData[];
}
```

Finalization Details and Pitfalls

```
img = new RGBImage1();
```




```
img = null; and after a subsequent GC...
```



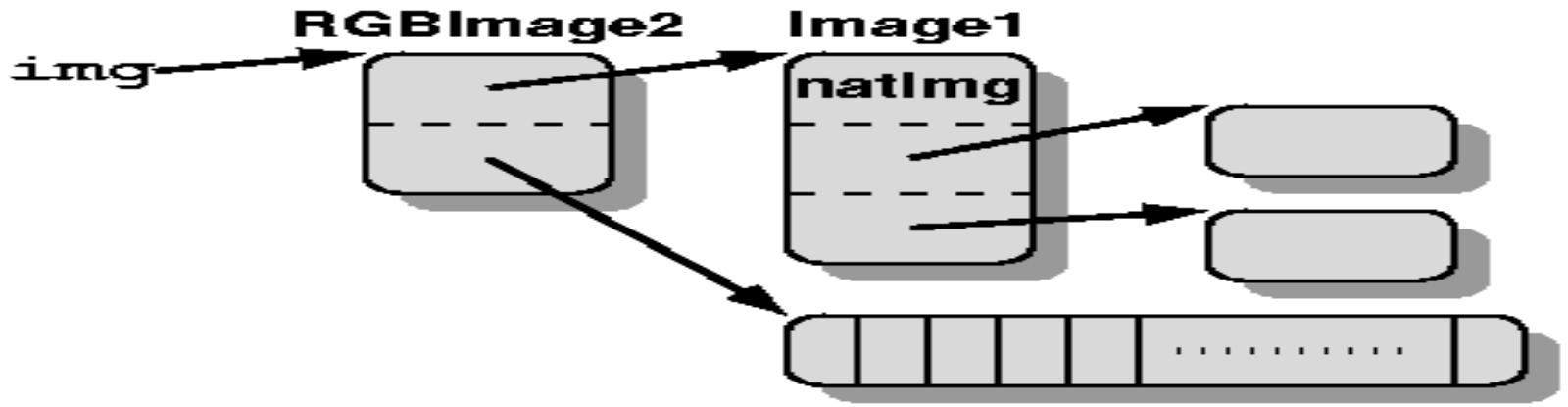
Finalization Details and Pitfalls

- One way to avoid this problem is to rearrange the code so that it uses composition instead of inheritance, as follows

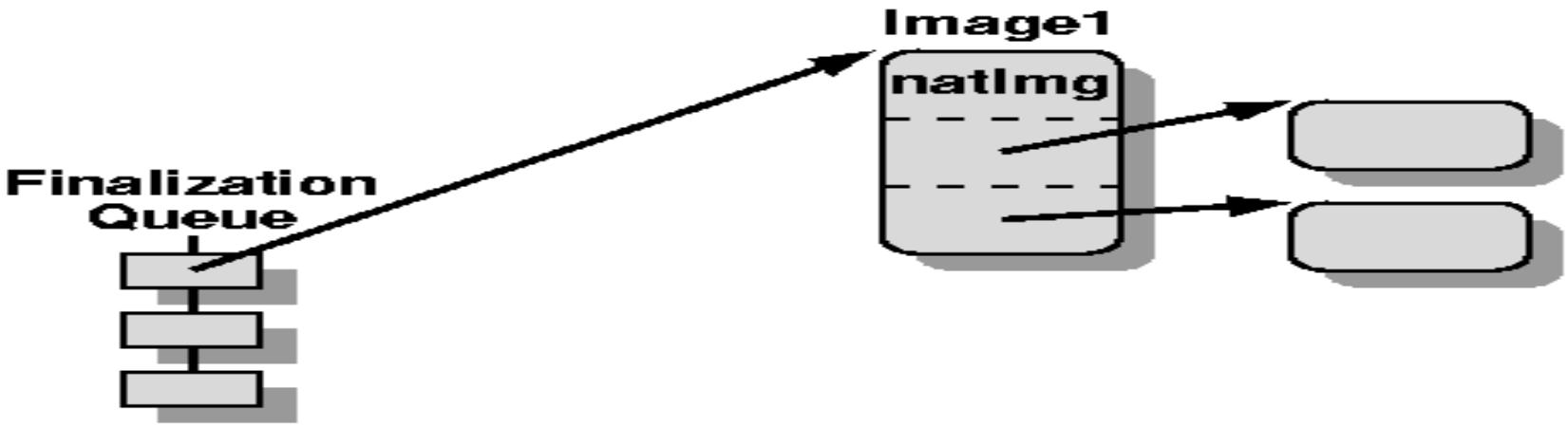
```
public class RGBImage2 {
        private Image1 img;
        private byte rgbData[];

        public void dispose() {
                img.dispose();
        }
}
```

Finalization Details and Pitfalls

```
img = new RGBImage2();
```




```
img = null; and after a subsequent GC...
```



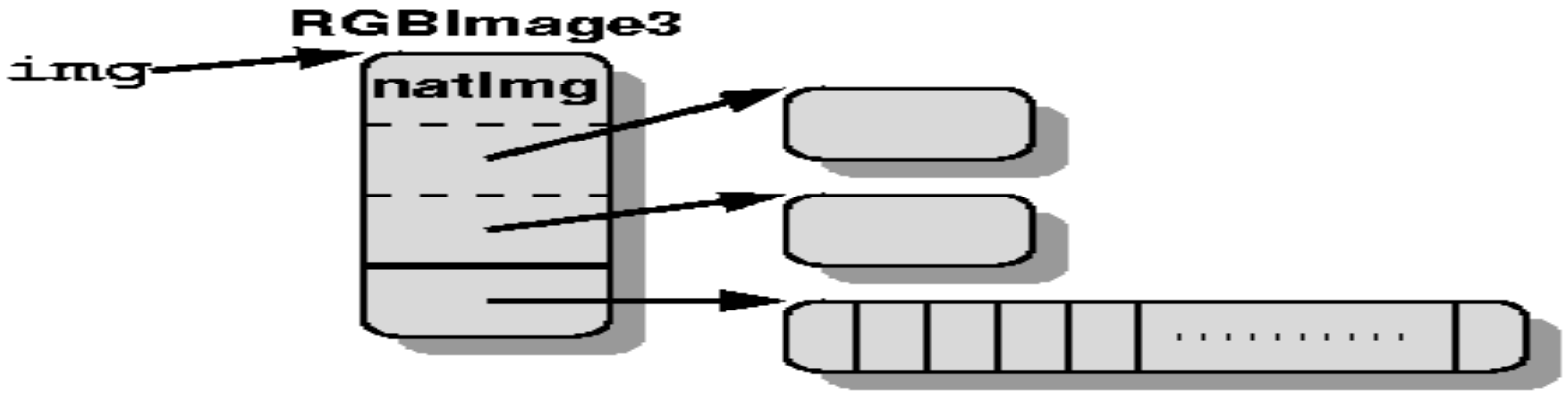
Finalization Details and Pitfalls

- You cannot always rearrange your code in the manner just described, however. Sometimes, as a user of the class, you will have to do more work to ensure that its instances do not hold on to more space than necessary when they are being finalized.

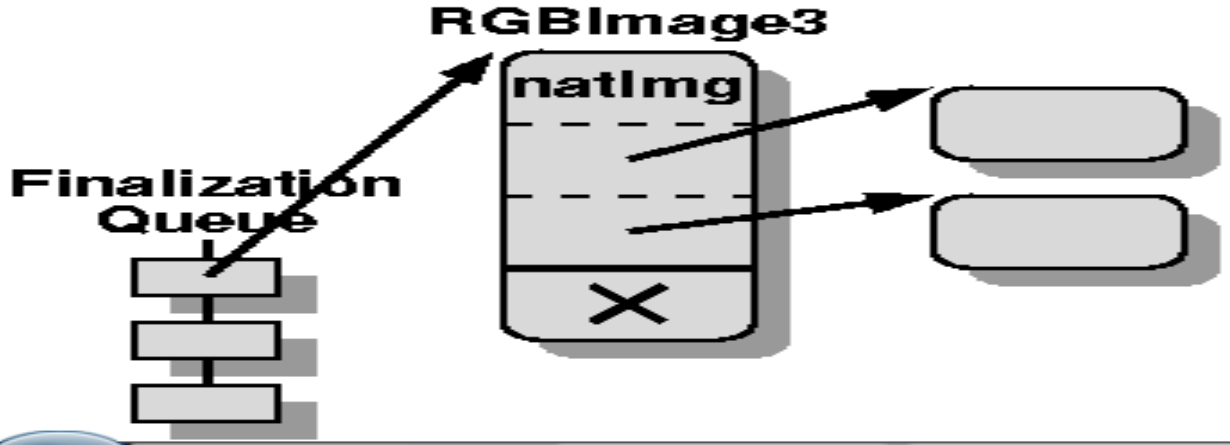
```
public class RGBImage3 extends Image1 {
        private byte rgbData[];

        public void dispose() {
                rgbData = null;
                super.dispose();
        }
}
```

Finalization Details and Pitfalls

```
img = new RGBImage3();
```




```
img.dispose(); img = null;
```
and after a subsequent GC...



Writing Classes to avoid finalize pitfalls

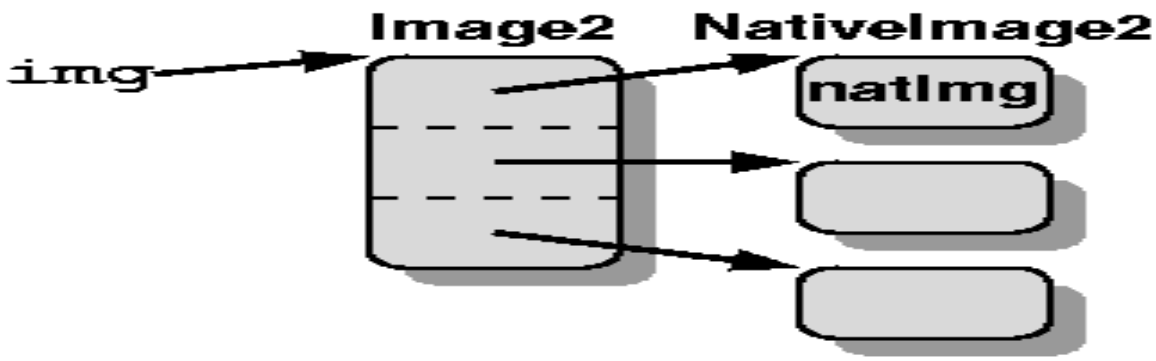
- We have described how to avoid memory-retention problems when working with third-party classes that use finalizers.
- Now let's look at how to write classes that require postmortem cleanup so that their users do not encounter the problems previously outlined.
- The best way to do so is to split such classes into two -- one to hold the data that need postmortem cleanup, the other to hold everything else -- and define a finalizer only on the former

Writing Classes to avoid finalize pitfalls

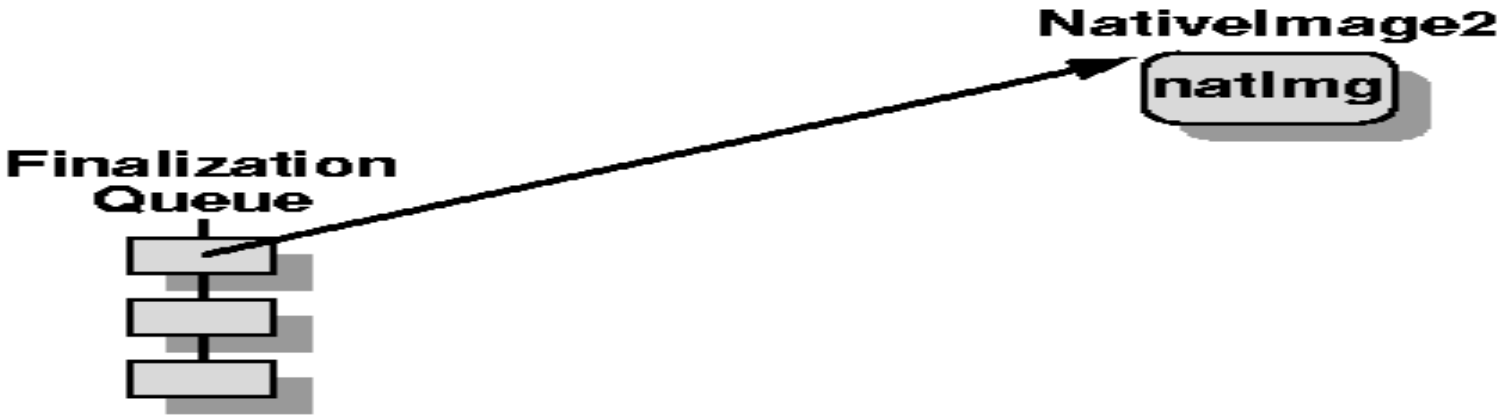
```
final class NativeImage2 {
        // pointer to the native image data
        private int nativeImg;

        // it disposes of the native image;
        // successive calls to it will be ignored
        private native void disposeNative();
        void dispose() { disposeNative(); }
        protected void finalize() { dispose(); }
}

public class Image2 {
        private NativeImage2 nativeImg;
        private Point pos;
        private Dimension dim;

        public void dispose() { nativeImg.dispose(); }
}
```

Writing Classes to avoid finalize pitfalls

```
img = new Image2();
```




```
img = null;
```
and after a subsequent GC...



Alternative to finalize: Weak References

- Unless you know exactly when the socket is no longer needed by the program and remember to remove the corresponding mapping from the Map, the Socket and User objects will stay in the Map forever.

```
public class SocketManager {
    private Map<Socket,User> m = new HashMap<Socket,User>();

    public void setUser(Socket s, User u) {
        m.put(s, u);
    }
    public User getUser(Socket s) {
        return m.get(s);
    }
    public void removeUser(Socket s) {
        m.remove(s);
    }
}

SocketManager socketManager;
...
socketManager.setUser(socket, user);
```

Alternative to finalize: Weak References

- Fortunately, as of JDK 1.2, the garbage collector provides a means to declare such object lifecycle dependencies so that the garbage collector can help us prevent this sort of memory leak -- with *weak references*.
- With weak references, you can maintain a reference to the referent without preventing it from being garbage collected.
- When the garbage collector traces the heap, if the only outstanding references to an object are weak references, then the referent becomes a candidate for GC as if there were no outstanding references, and any outstanding weak references are *cleared*.

Alternative to finalize: Weak References

- If the weak reference has been cleared (either because the referent has already been garbage collected or because someone called WeakReference.clear()), get() returns null.
- Accordingly, you should always check that get() returns a non-null value before using its result, as it is expected that the referent will eventually be garbage collected.

Alternative to finalize: Weak References

```
public class WeakHashMap<K,V> implements Map<K,V> {

    private static class Entry<K,V> extends WeakReference<K>
      implements Map.Entry<K,V> {
        private V value;
        private final int hash;
        private Entry<K,V> next;
        ...
    }

    public V get(Object key) {
        int hash = getHash(key);
        Entry<K,V> e = getChain(hash);
        while (e != null) {
            K eKey= e.get();
            if (e.hash == hash && (key == eKey || key.equals(eKey)))
                return e.value;
            e = e.next;
        }
        return null;
    }
```

Alternative to finalize: Weak References

- **Reference queues**
  - WeakHashMap uses weak references for holding map keys, which allows the key objects to be garbage collected when they are no longer used by the application, and the get() implementation can tell a live mapping from a dead one by whether WeakReference.get() returns null.
  - But this is only half of what is needed to keep a Map's memory consumption from increasing throughout the lifetime of the application; something must also be done to prune the dead entries from the Map after the key object has been collected.

Alternative to finalize: Weak References

```
private void expungeStaleEntries() {
    Entry<K,V> e;
    while ( (e = (Entry<K,V>) queue.poll()) != null) {
        int hash = e.hash;

        Entry<K,V> prev = getChain(hash);
        Entry<K,V> cur = prev;
        while (cur != null) {
            Entry<K,V> next = cur.next;
            if (cur == e) {
                if (prev == e)
                    setChain(hash, next);
                else
                    prev.next = next;
                break;
            }
            prev = cur;
            cur = next;
        }
    }
}
```

Alternative to finalize: Weak References

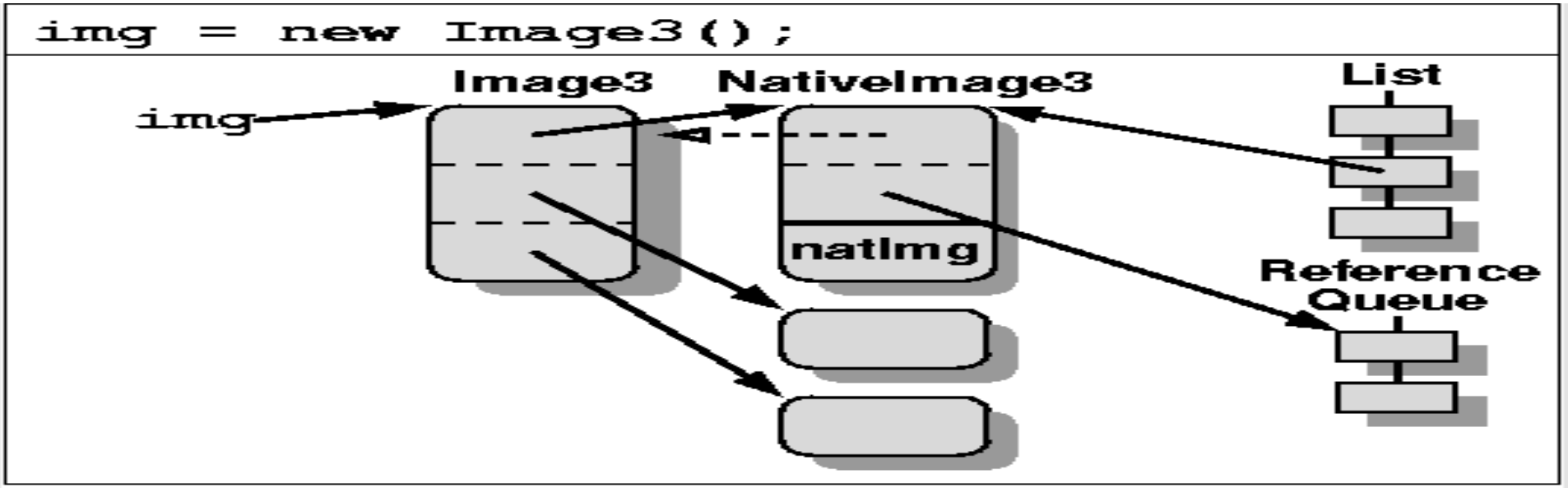
- Fixing the leak in SocketManager is easy; simply replace the HashMap with a WeakHashMap.
- You can use this approach whenever the lifetime of the mapping must be tied to the lifetime of the key.

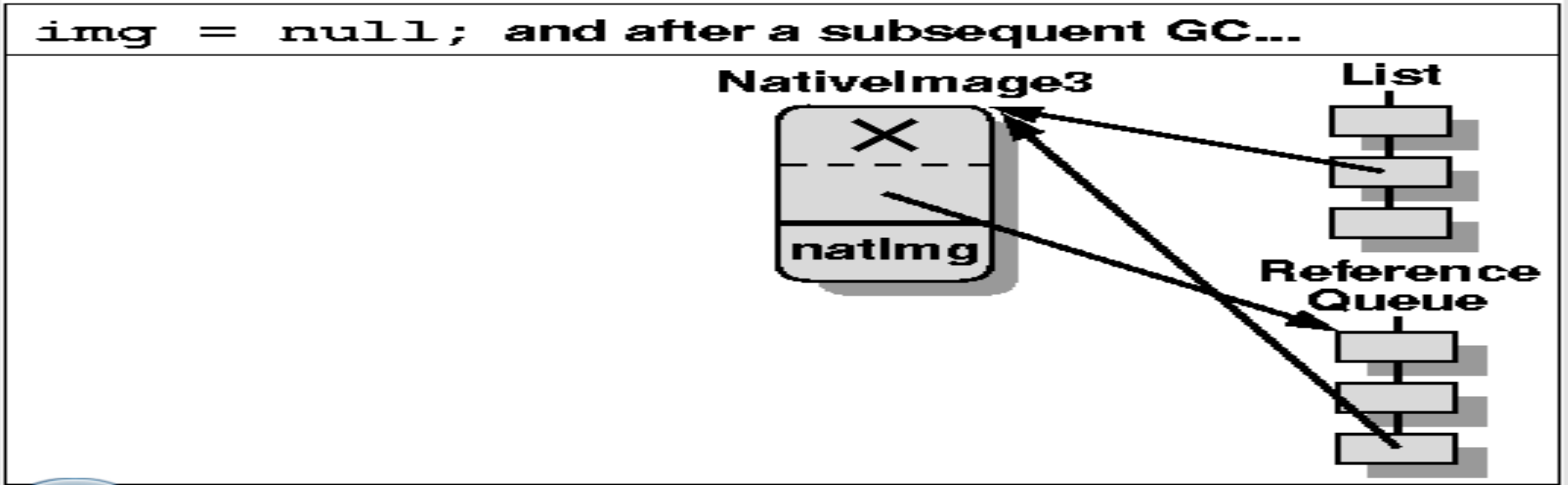
```
public class SocketManager {
    private Map<Socket,User> m = new WeakHashMap<Socket,User>();

    public void setUser(Socket s, User u) {
        m.put(s, u);
    }
    public User getUser(Socket s) {
        return m.get(s);
    }
}
```

Weak References addressing cleanups

```
final class NativeImage3 extends WeakReference<Image3> {
        // pointer to the native image data
        private int nativeImg;

        // it disposes of the native image;
        // successive calls to it will be ignored
        private native void disposeNative();
        void dispose() {
                refList.remove(this);
                disposeNative();
        }

        static private ReferenceQueue<Image3> refQueue;
        static private List<NativeImage3> refList;
        static ReferenceQueue<Image3> referenceQueue() {
                return refQueue;
        }

        NativeImage3(Image3 img) {
                super(img, refQueue);
                refList.add(this);
        }
}

public class Image3 {
        private NativeImage3 nativeImg;
        private Point pos;
        private Dimension dim;

        public void dispose() { nativeImg.dispose(); }
}
```

Weak References addressing cleanups





Weak References addressing cleanups

- You can do this with the following method, executed, say, on a "cleanup" thread:

```
static void drainRefQueueLoop() {
        ReferenceQueue<Image3> refQueue =
                NativeImage3.referenceQueue();
        while (true) {
                NativeImage3 nativeImg =
                        (NativeImage3) refQueue.remove();
                nativeImg.dispose();
        }
}
```

Weak References addressing cleanups

- There are cases, however, in which it might not be easy or desirable to introduce a new thread in an application. In such cases, an alternative is to drain the reference queue before every NativeImage3 instance allocation.

```
static final private int MAX_ITERATIONS = 2;
static void drainRefQueueBounded() {
        ReferenceQueue<Image3> refQueue =
                NativeImage3.referenceQueue();
        int iterations = 0;
        NativeImage3 nativeImg = (NativeImage3) refQueue.poll();
        while (nativeImg != null && iterations < MAX_ITERATIONS) {
                nativeImg.dispose();
                ++iterations;
                nativeImg = (NativeImage3) refQueue.poll();
        }
}
```

Weak References addressing cleanups

- **Weak references and Phantom references are eagerly created so do NOT use them for caches.**
  - They are collected at minor collections.

Soft references

- **Programmatic Behavior is the same as Weak reference but they are collected lazily.**
  - Now, SoftReferences remain alive for some time after the last time they were referenced. The default length of time value is one second of lifetime per free megabyte in the heap. This provides more of a differentiation between SoftReference and WeakReference behavior.
  - The initial time-to-live values for SoftReferences can be altered using the -XX:SoftRefLRUPolicyMSPerMB flag, which specifies the lifetime per free megabyte in the heap, in milliseconds. To change the value to 3 seconds per free heap megabyte, you would use:
  - java -XX:SoftRefLRUPolicyMSPerMB=3000 ...

Soft references

- If you need to cache more than a single object, you might use a Map, but you have a choice as to how to employ soft references. You could manage the cache as a Map<K, SoftReference<V>> or as a SoftReference<Map<K,V>>.
- The latter option is usually preferable, as it makes less work for the collector and allows the entire cache to be reclaimed with less effort when memory is in high demand.

LOGO

Thank You !