Understanding, Using, and Debugging Java Reference Objects

Important Disclaimers

THE INFORMATION CONTAINED IN THIS PRESENTATION IS PROVIDED FOR INFORMATIONAL PURPOSES ONLY.

WHILST EFFORTS WERE MADE TO VERIFY THE COMPLETENESS AND ACCURACY OF THE INFORMATION CONTAINED IN THIS PRESENTATION, IT IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED.

ALL PERFORMANCE DATA INCLUDED IN THIS PRESENTATION HAVE BEEN GATHERED IN A CONTROLLED ENVIRONMENT. YOUR OWN TEST RESULTS MAY VARY BASED ON HARDWARE, SOFTWARE OR INFRASTRUCTURE DIFFERENCES.

ALL DATA INCLUDED IN THIS PRESENTATION ARE MEANT TO BE USED ONLY AS A GUIDE.

IN ADDITION, THE INFORMATION CONTAINED IN THIS PRESENTATION IS BASED ON IBM'S CURRENT PRODUCT PLANS AND STRATEGY, WHICH ARE SUBJECT TO CHANGE BY IBM, WITHOUT NOTICE.

IBM AND ITS AFFILIATED COMPANIES SHALL NOT BE RESPONSIBLE FOR ANY DAMAGES ARISING OUT OF THE USE OF, OR OTHERWISE RELATED TO, THIS PRESENTATION OR ANY OTHER DOCUMENTATION.

NOTHING CONTAINED IN THIS PRESENTATION IS INTENDED TO, OR SHALL HAVE THE EFFECT OF:

- CREATING ANY WARRANT OR REPRESENTATION FROM IBM, ITS AFFILIATED COMPANIES OR ITS OR THEIR SUPPLIERS AND/OR LICENSORS

Introduction to the speaker

- Many years experience Java development
  - IBM CICS and Java technical consultant
  - WAS & CICS integration using Java
  - IBM WebSphere sMash PHP implementation
  - CICS Dynamic Scripting
  - IBM Memory Analyzer

- Recent work focus:
  - IBM Monitoring and Diagnostic Tools for Java
  - Eclipse Memory Analyzer (MAT) project

- My contact information:
  - Contact me through the Java Technology Community on developerWorks (JonathanPLawrence).

Understanding, Using & Debugging Java References

- Some history , visiting finalizers, some basics about GC

- The principles of Java references

- Simple guidelines to usage – do's and don'ts etc.

- Using Memory Analyzer to help resolve related problems

- Wrap-up - caveat emptor , your mileage may vary, here be dragons...

First, some history

Way back, in the dawn of Java history, it became evident that there was a real need to know when an object was going to be garbage collected...

And so was born the idea of a "finalizer"

Oh Dear!

Finalizers – good idea, fundamental flaws...

Somehow the idea of "tell **me** when an object has been collected"

Turned into:

"tell the **object** itself that it is eligible to be collected"

Oh Dear!

# Why are finalizers bad for your health?

To answer this question we need to look at the fundamentals of an object oriented, garbage collected programming environment

At its simplest, garbage collection scans the heap to find and remove objects that cannot be reached from following external "root" references

Think of it a bit like a combination of a banana tree, gravity and a monkey...

+ Newton + = GC
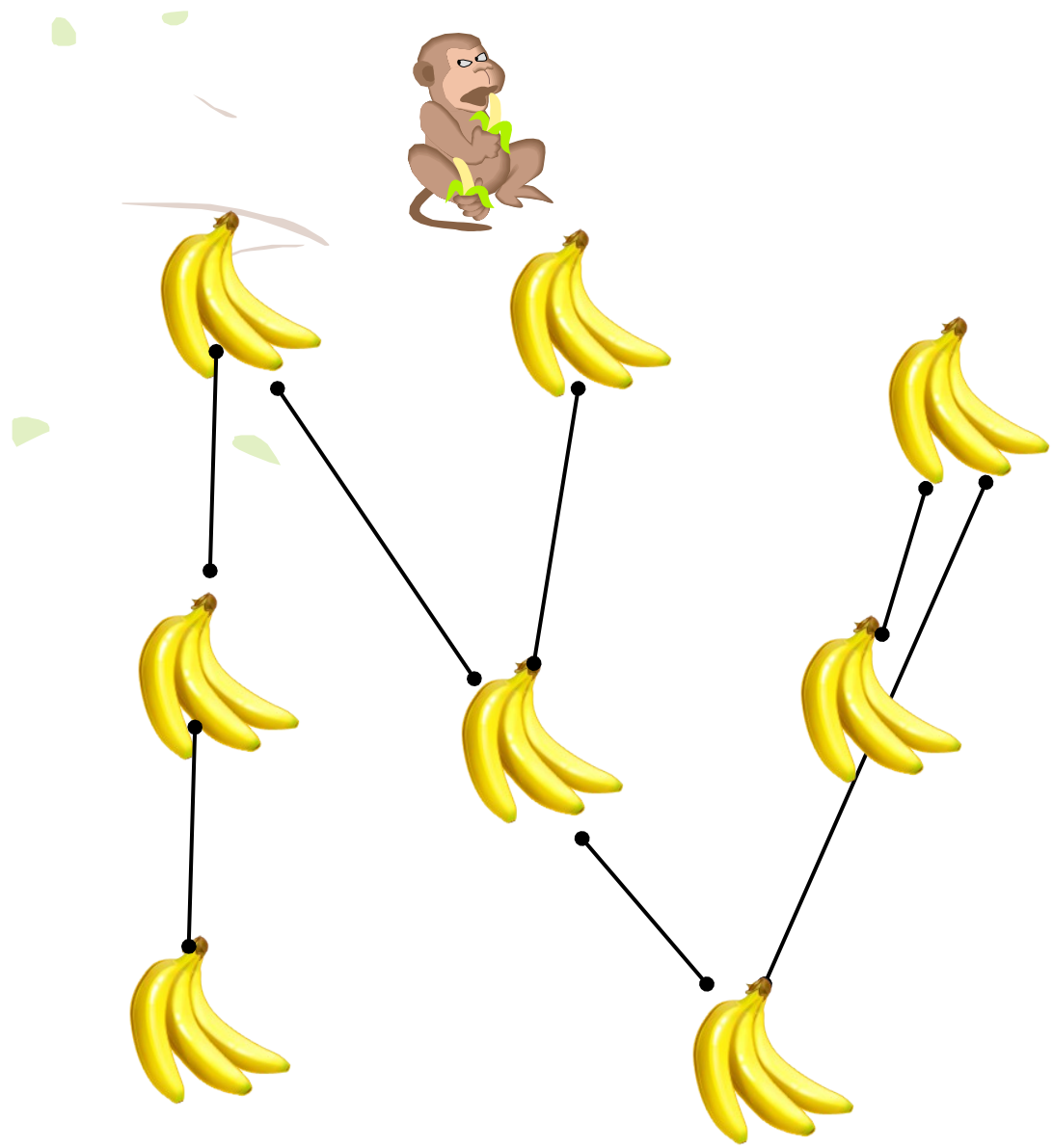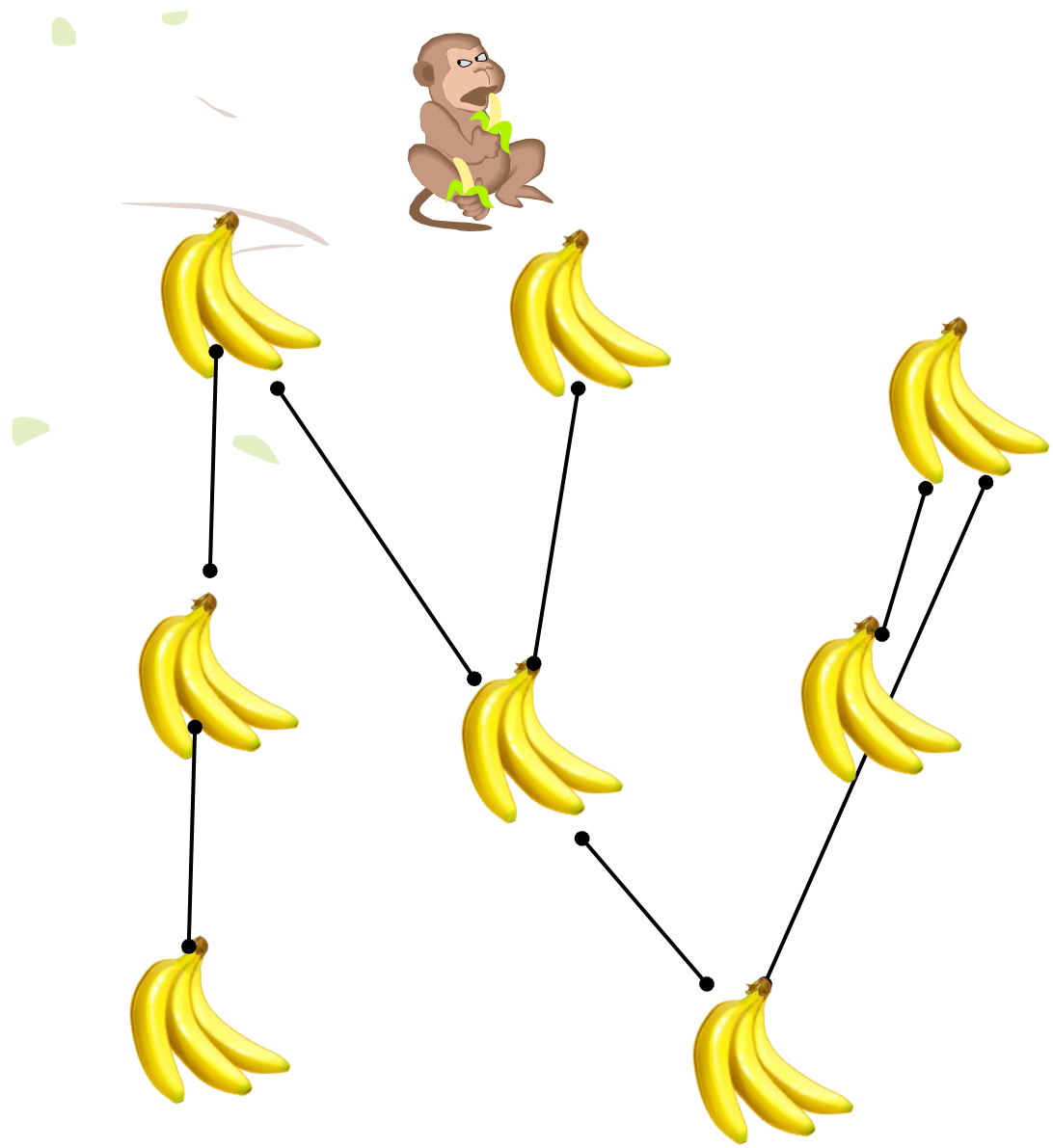
That was Garbage Collection at its simplest – but let's recap

GC visits each Root object and follows all references to other objects.

Each newly visited object is **marked** and it's object references are followed

Any object not marked is **unreachable** and is removed

See – simple…

Lets add finalizers…

GC with finalizers

GC visits each Root object and follows all references to other objects.

Each newly visited object is **marked** and it's object references are followed

Any object not marked but **eligible for finalization** is **marked** and added to the finalization queue. **All of its references are visited and marked**

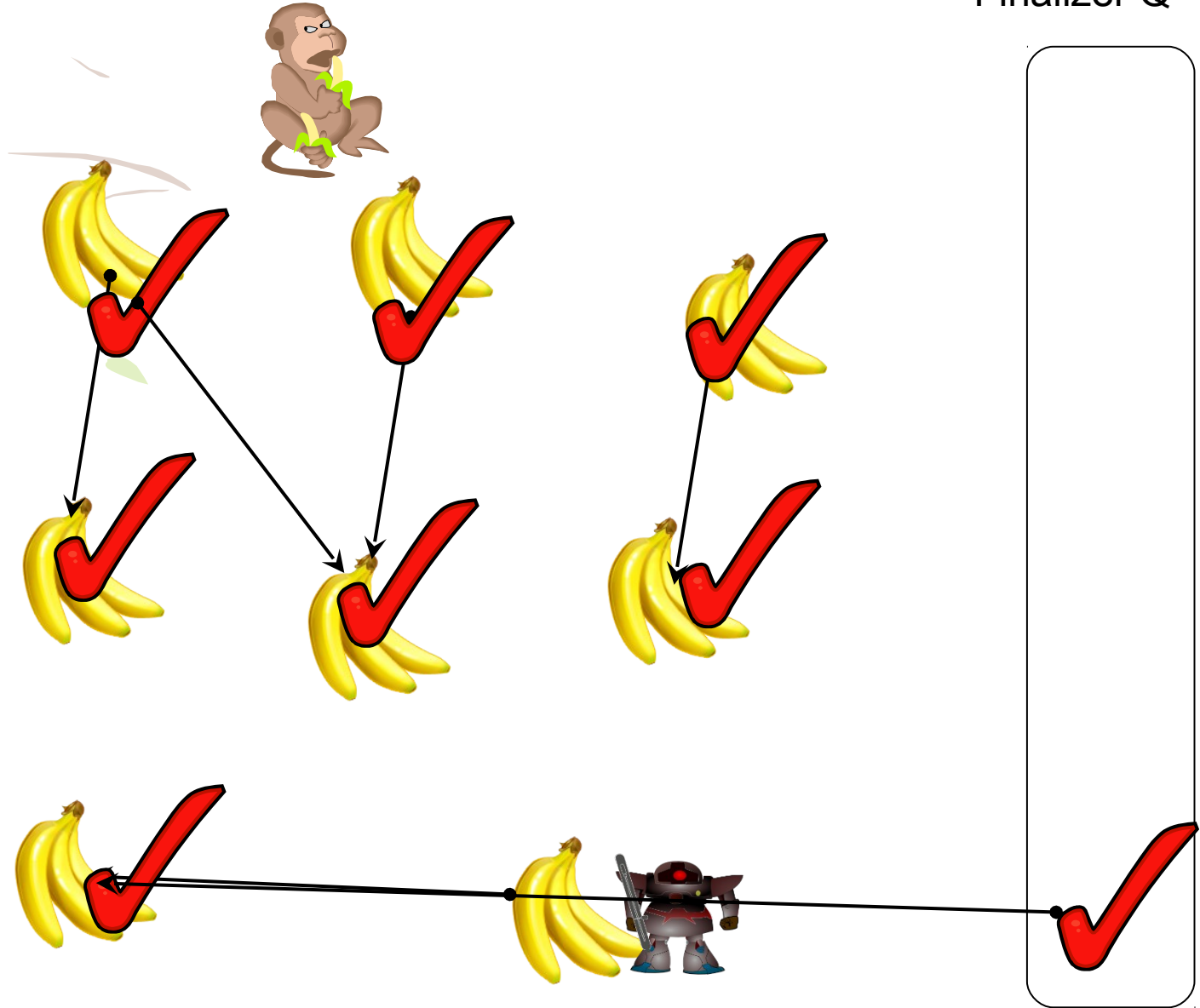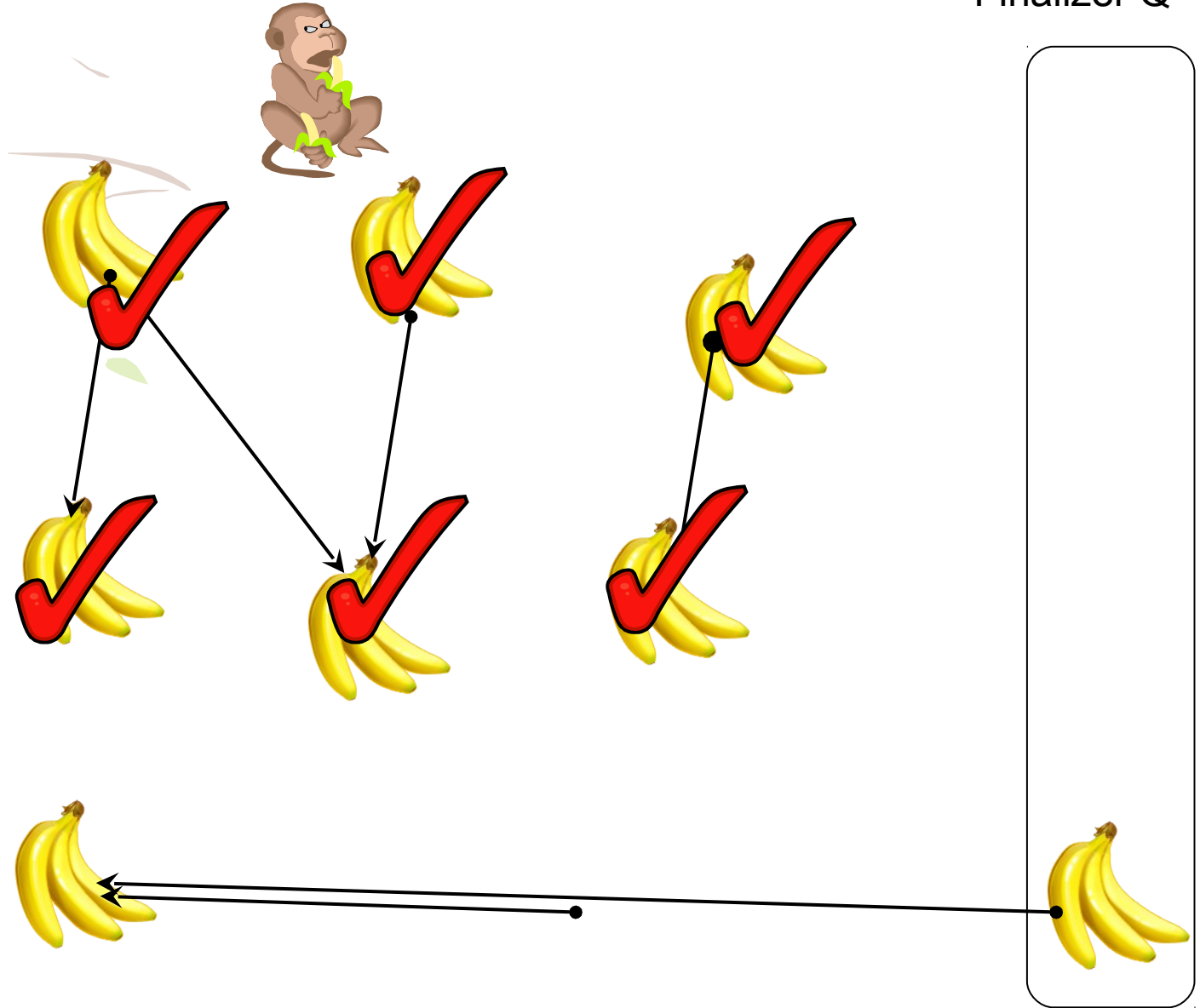Any object not marked is **unreachable** and is removed
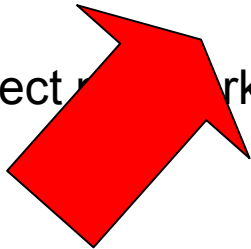
Oh Dear!

Finalizer Q

GC: collected 0

Any object not marked but **eligible for finalization** is **marked** and added to the finalization queue. **All of its references are visited and marked**

© 2011, 2012 IBM Corporation

GC: collected 0

Finalizer runs

GC: collected 2

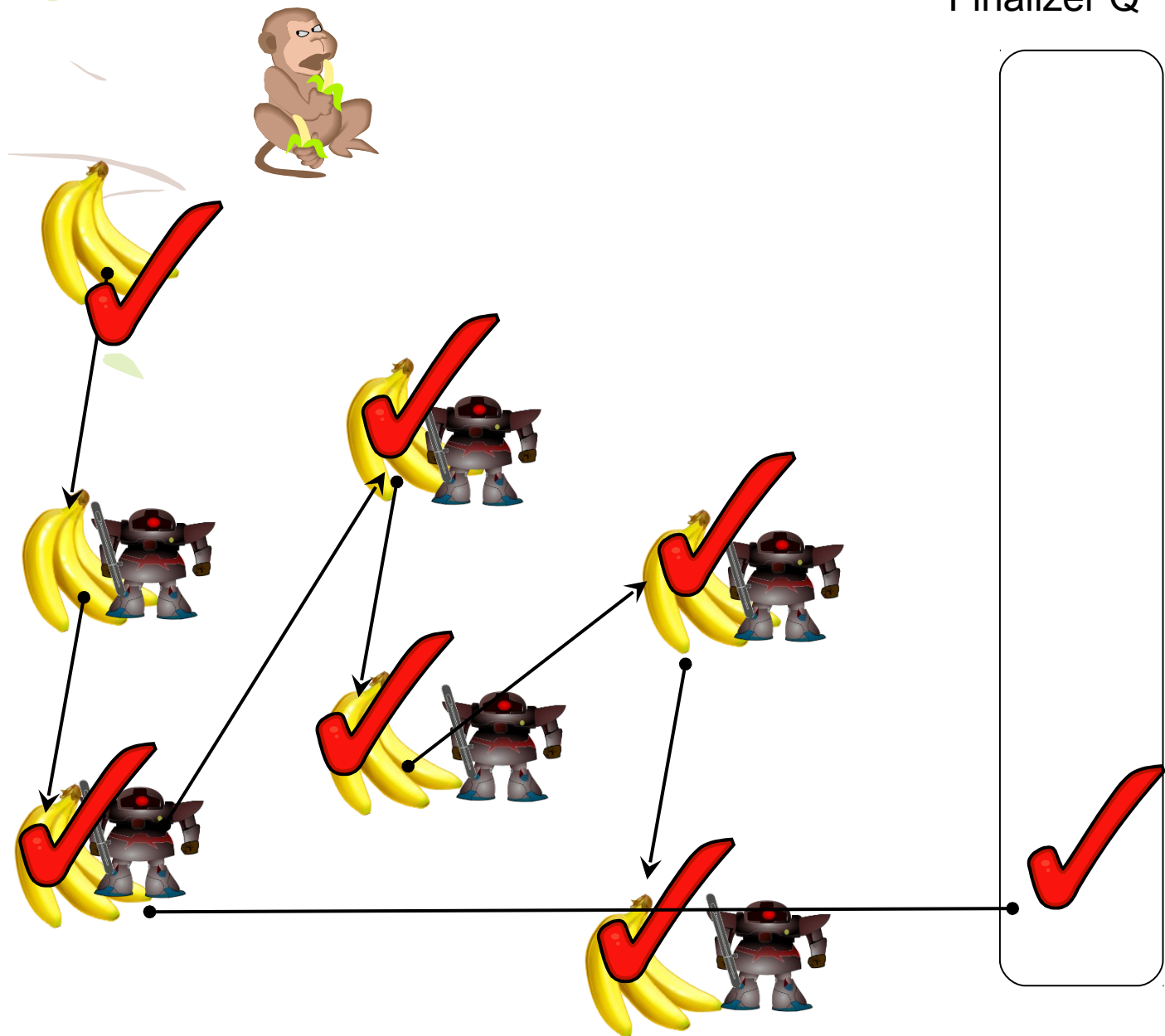Any object not marked but **eligible for finalization** is **marked** and added to the finalization queue. **All of its references are visited and marked**

That looked easy - using finalizers just delays
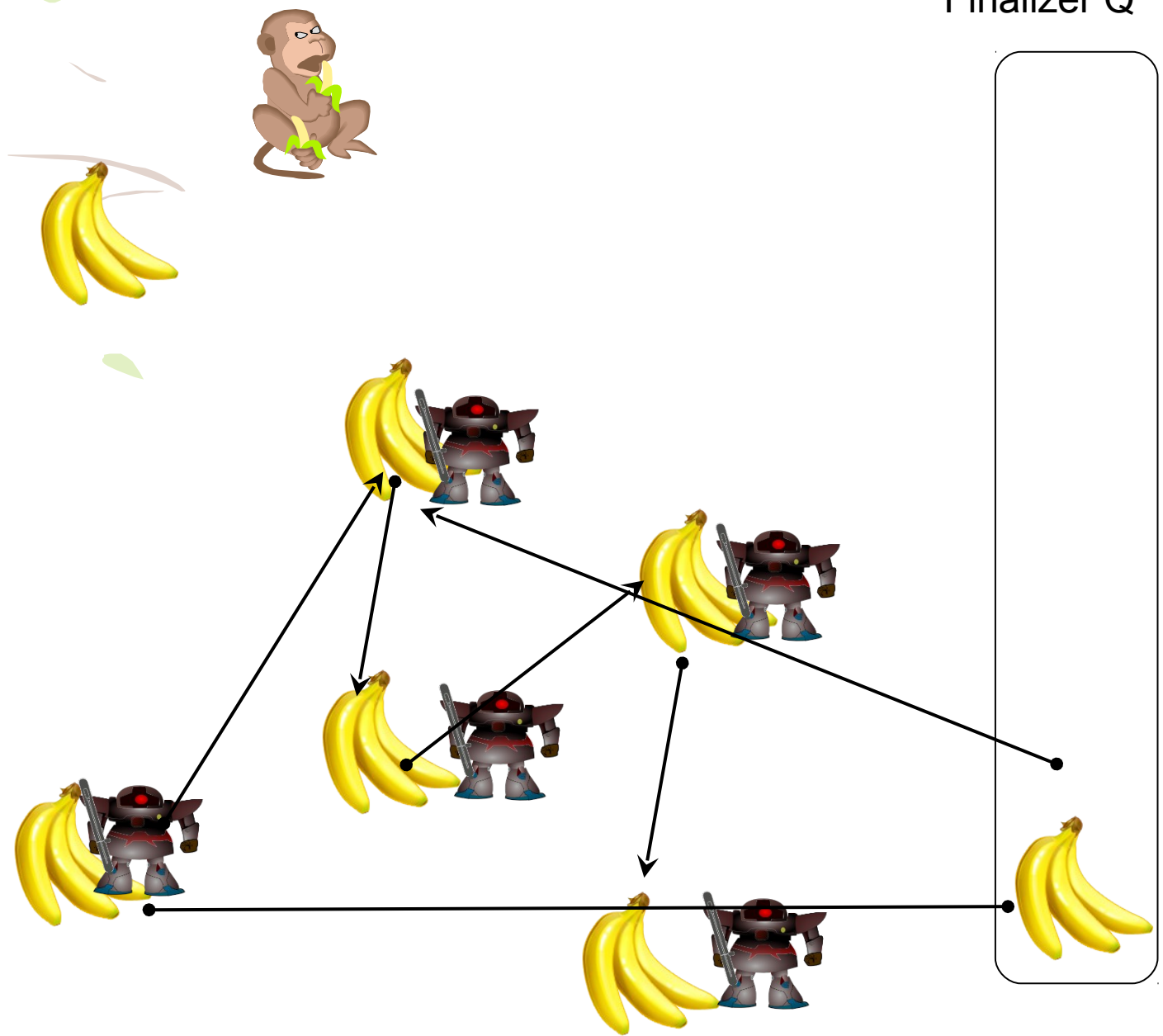the object collection. Is that it?

GC: collected 0

GC: collected 0

Finalizer runs

GC: collected 1
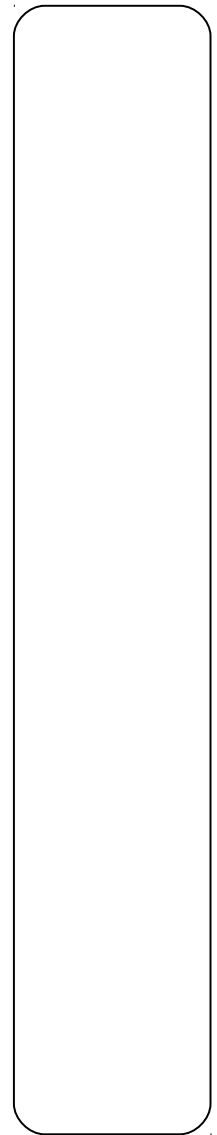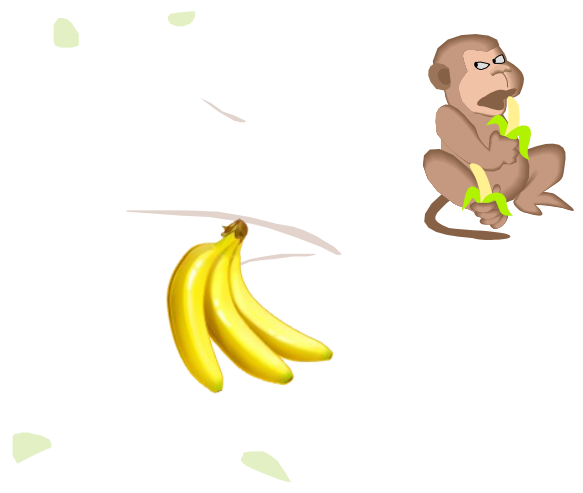
GC: collected 0

Finalizer runs

GC: collected 1

GC: collected 0

Finalizer runs

GC: collected 1

Finalizer runs

GC: collected 1

Finalizer runs

GC: collected 1

Finalizer runs

GC: collected 1

Finalizer runs

GC: collected 1

Finalizers are bad:

Used incorrectly they can cause Out of Memory conditions:
Inappropriate linkages between objects with finalizers
finalizer methods blocking when executed
finalizer methods taking too long to execute.

By the way:

The finalizer contract with the JVM says

   "The Java programming language does not guarantee which thread will invoke the finalize method"

In fact the JVM doesn't guarantee to run finalizers ever.
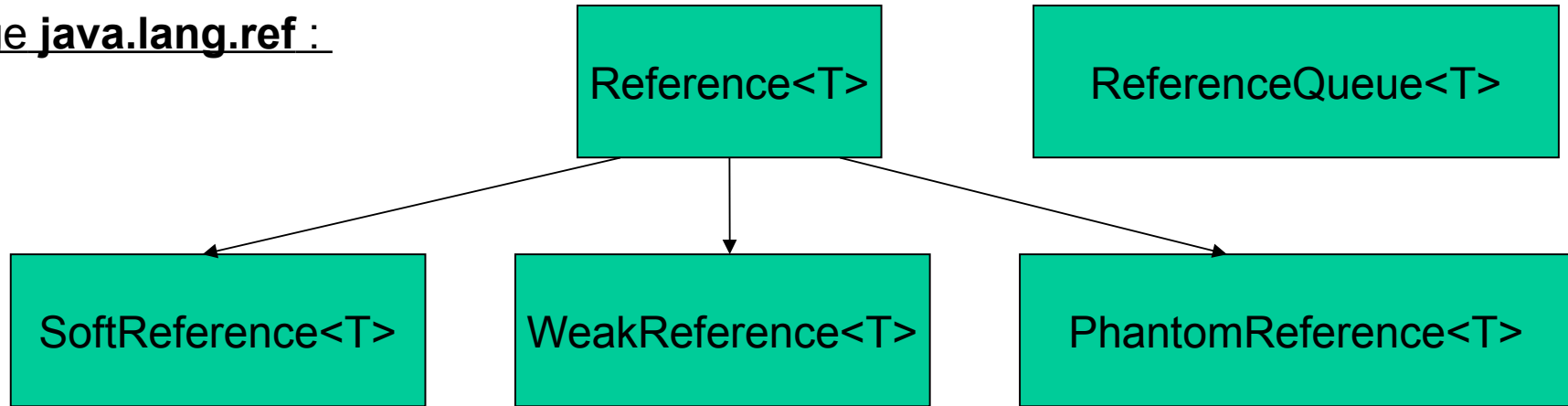
Convinced yet?

Onwards to Java References

The Java Reference API is intended to provide a mechanism to replace finalizers, and give slightly improved memory management facilities

Introducing Java Reference Objects

- Introduced in Java SE v1.2

package **java.lang.ref** :

Reference<T>		ReferenceQueue<T>

SoftReference<T>	WeakReference<T>	PhantomReference<T>

http://download.oracle.com/javase/7/docs/api/java/lang/ref/package-summary.html

package **java.util** :

WeakHashMap<K,V>

Your friend

http://download.oracle.com/javase/7/docs/api/index.html?java/util/WeakHashMap.html

Concept

Reference<T>

Create one
of these

To be the weak
link to this

foo

from this

bar
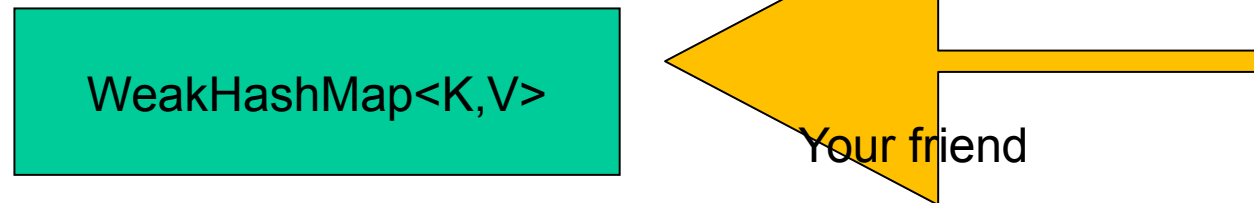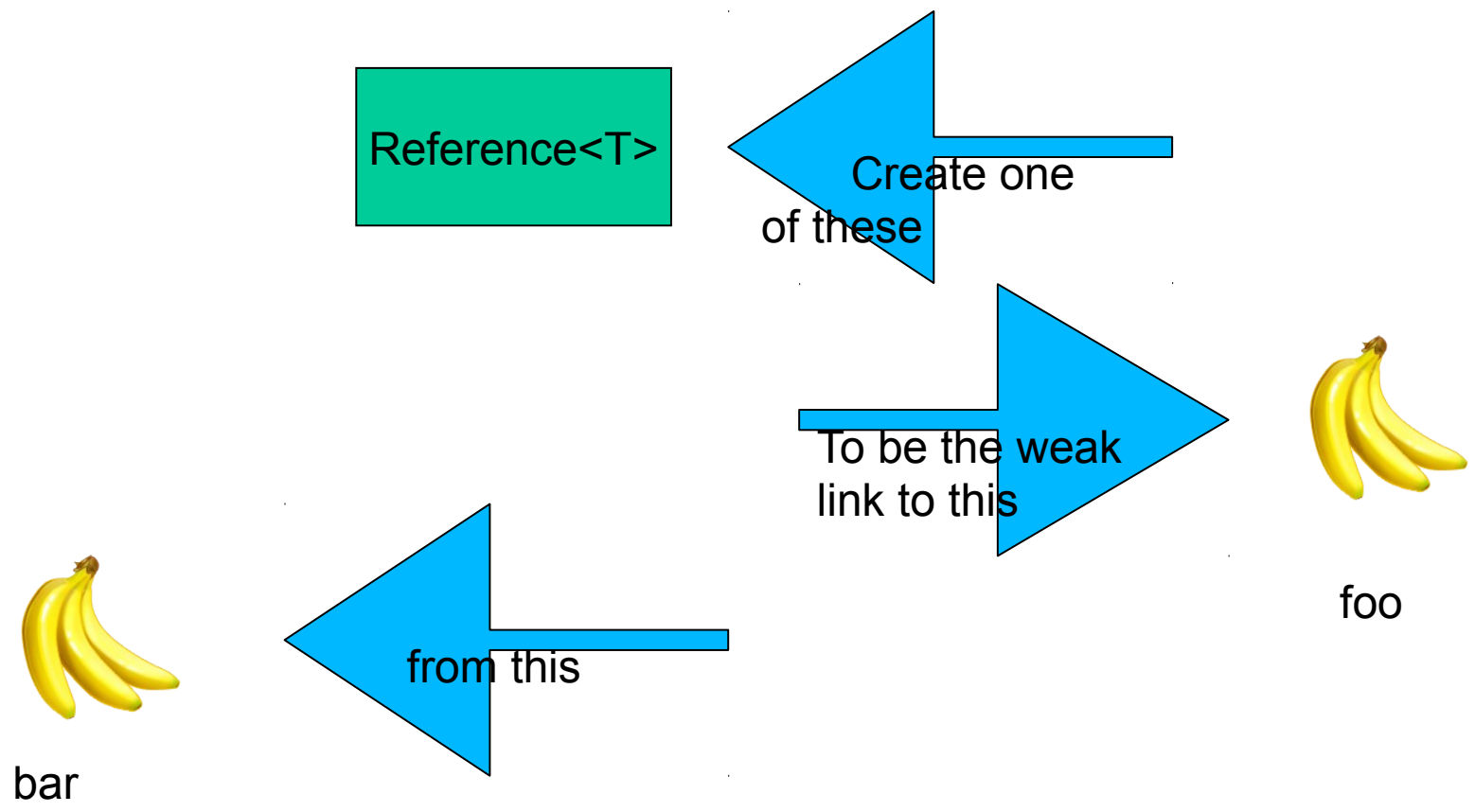
Reference r=new Reference(foo);

Concept

ReferenceQueue<T>

Use one
of these

To be informed
When this

foo

Reference<T>

Has gone
from this
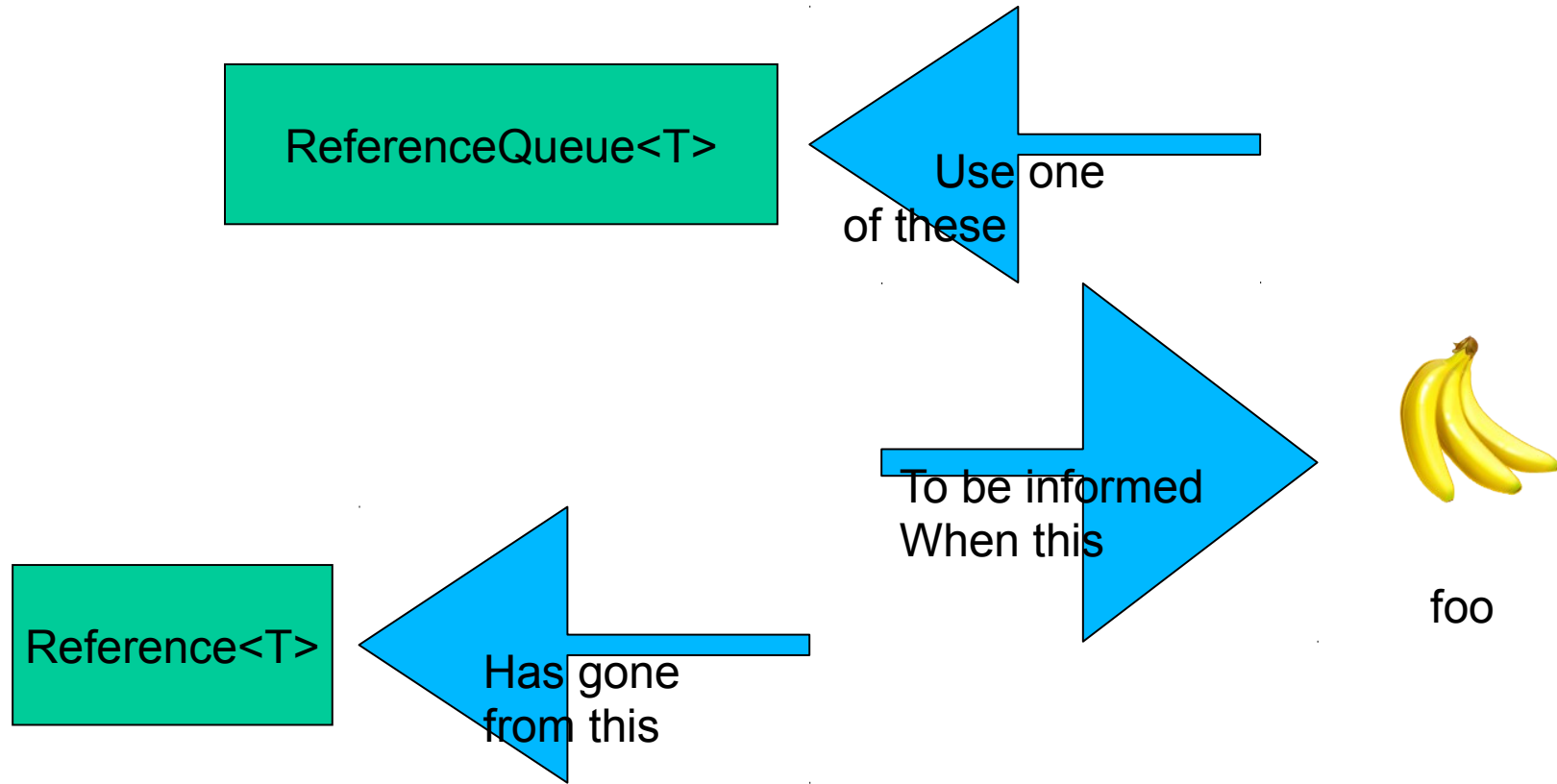
ReferenceQueue q=new ReferenceQueue();
Reference e=new Reference(foo,q);

This doesn't get queued – since there is a strong reference

Unlike finalizers - You have to process the reference q.
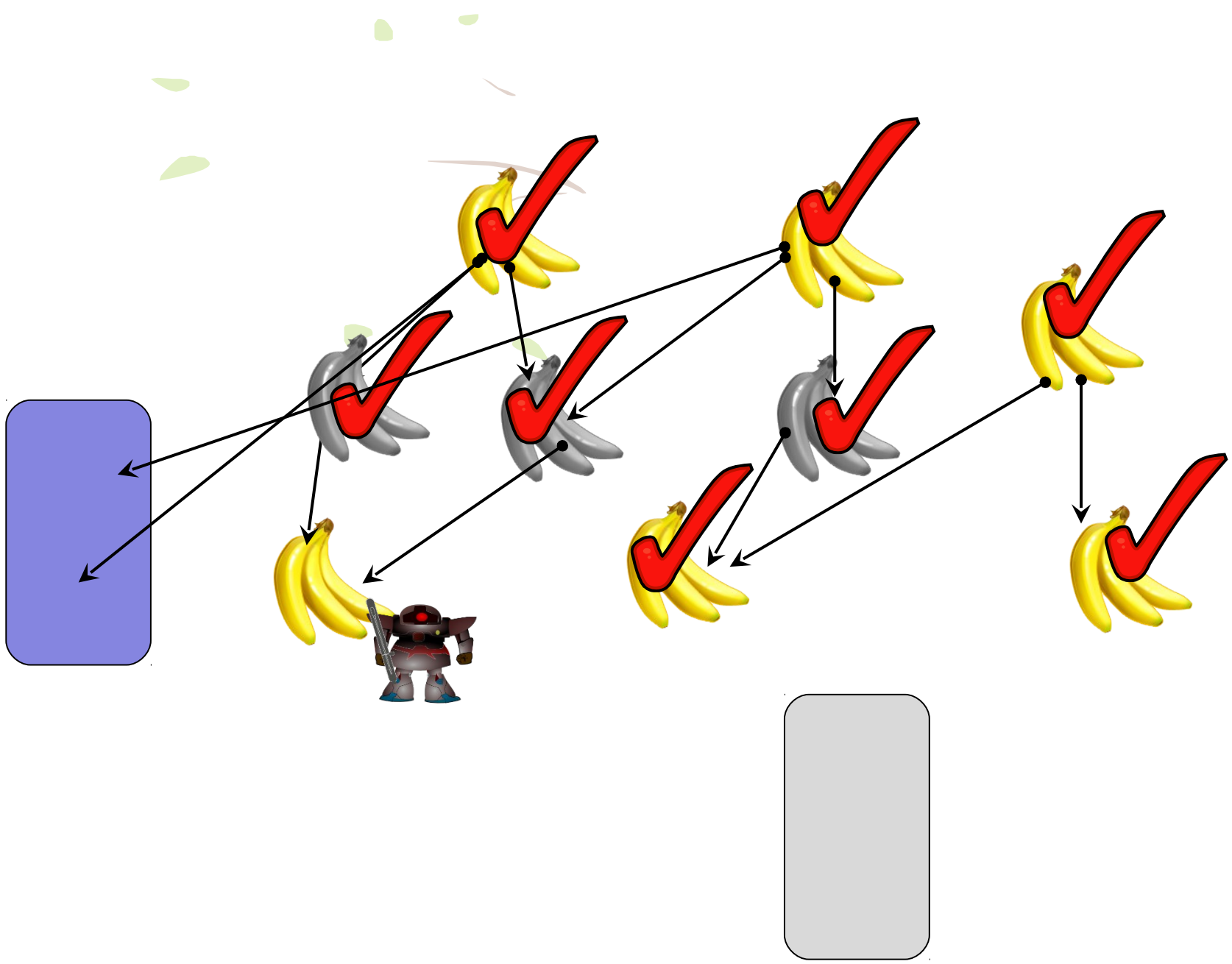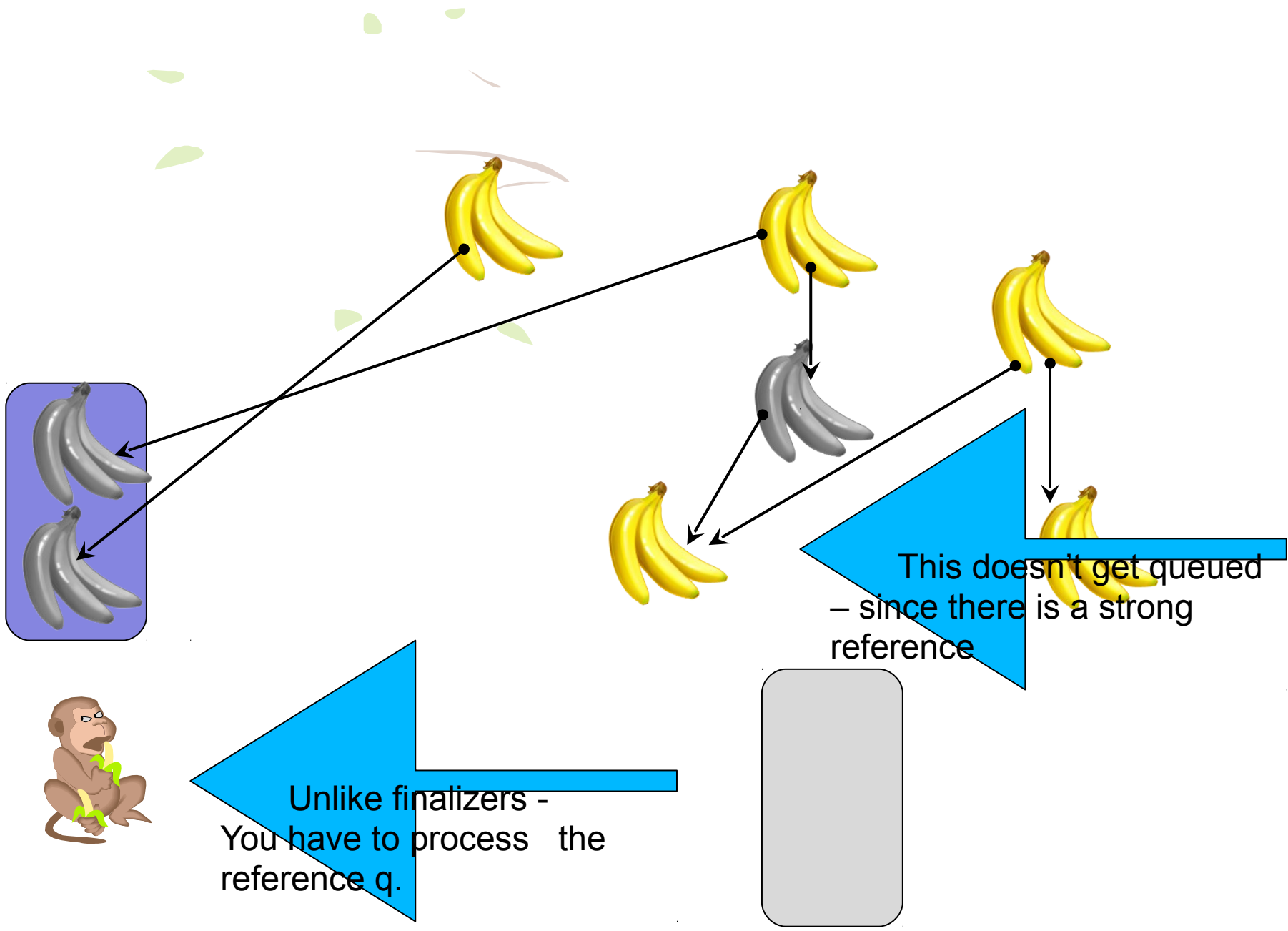
Reference Objects so far

A Reference Object provides a level of indirection between Java code and an object.

The encapsulated object (**referent**) may be garbage collected

JVM

```
Stack Variable  ──┐
                  ├──▶  Reference  ········▶  Referent
Heap Object  ─────┘        Object
```
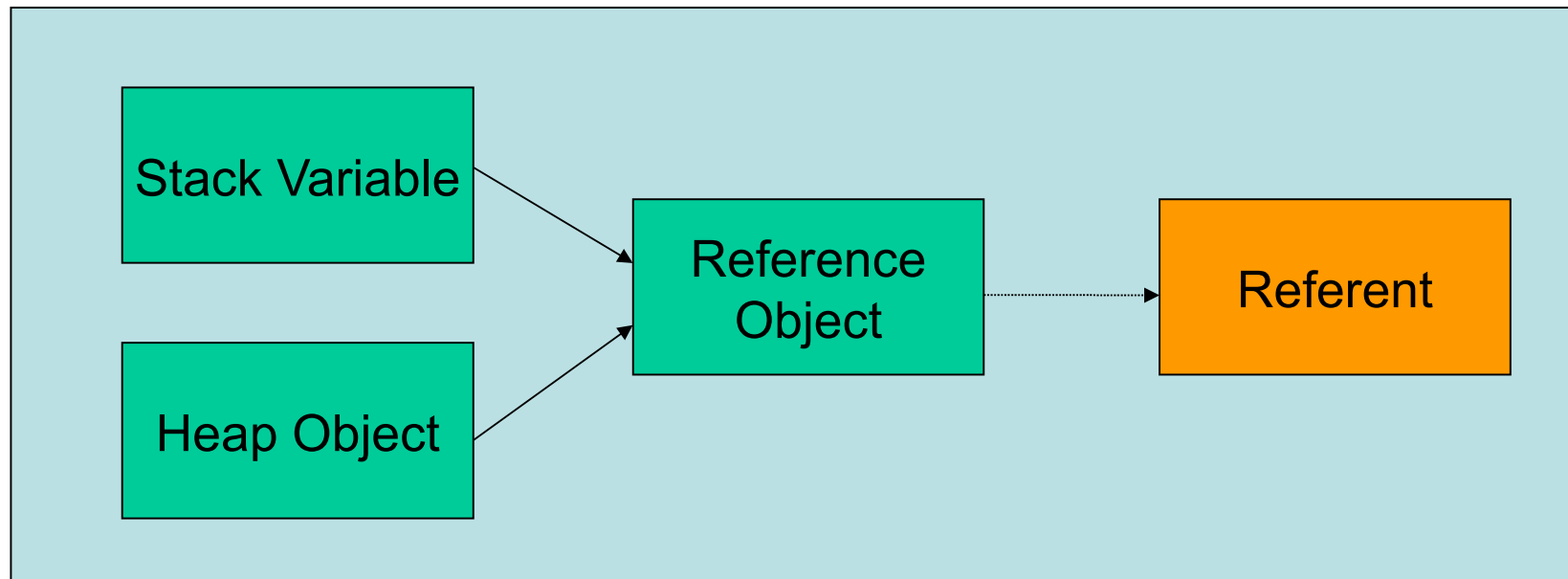
Referent is eligible for garbage collection unless a **strong** (normal) reference is kept elsewhere by the application.

Reference Object get() method returns referent unless it has been garbage collected.

Reference Queues

- A **ReferenceQueue** may optionally receive Reference Objects whose referents have been collected:

null <- poll() ← | ReferenceQueue | (empty)

obj <- get() ← | Reference | ┈┈> | Referent |
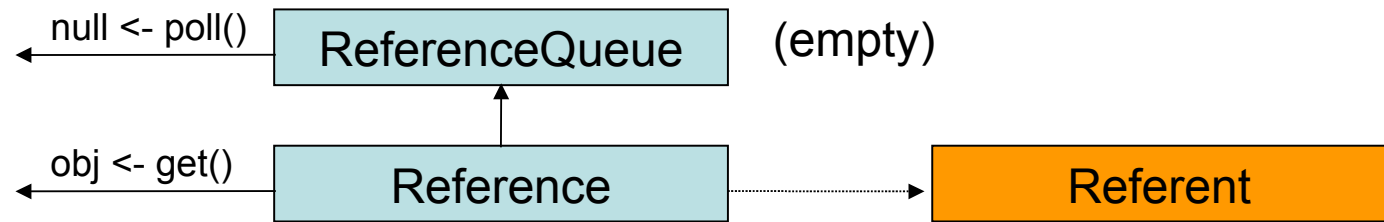
garbage collection
┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈┈

Reference Queues

- A **ReferenceQueue** may optionally receive Reference Objects whose referents have been collected:

null <- poll() ← **ReferenceQueue** (empty)

obj <- get() ← **Reference** ····> **Referent**

garbage collection
- -

No obj to return

null <- poll() ← **ReferenceQueue** (empty)

null <- get() ← **Reference** ··X··> **Referent**

Link to referent gone

Reference Queues

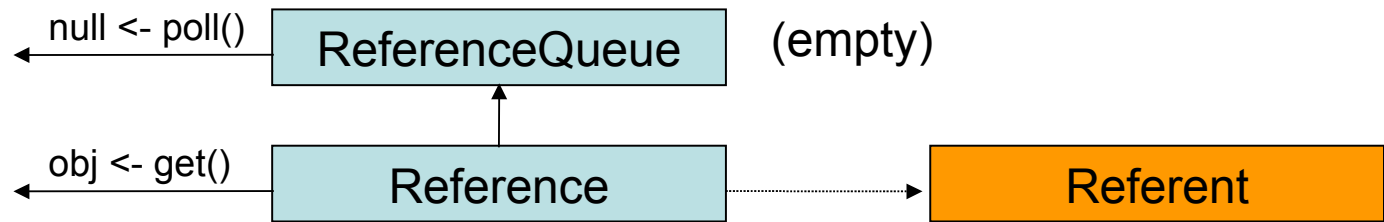- A **ReferenceQueue** may optionally receive Reference Objects whose referents have been collected:

null <- poll() → ReferenceQueue (empty)

obj <- get() → Reference ⟶ **Referent**

garbage collection

null <- poll() → ReferenceQueue (empty)

null <- get() → Reference   **Referent**

After garbage collection

Reference available on q

ref <- poll() → ReferenceQueue

null <- get() → Reference
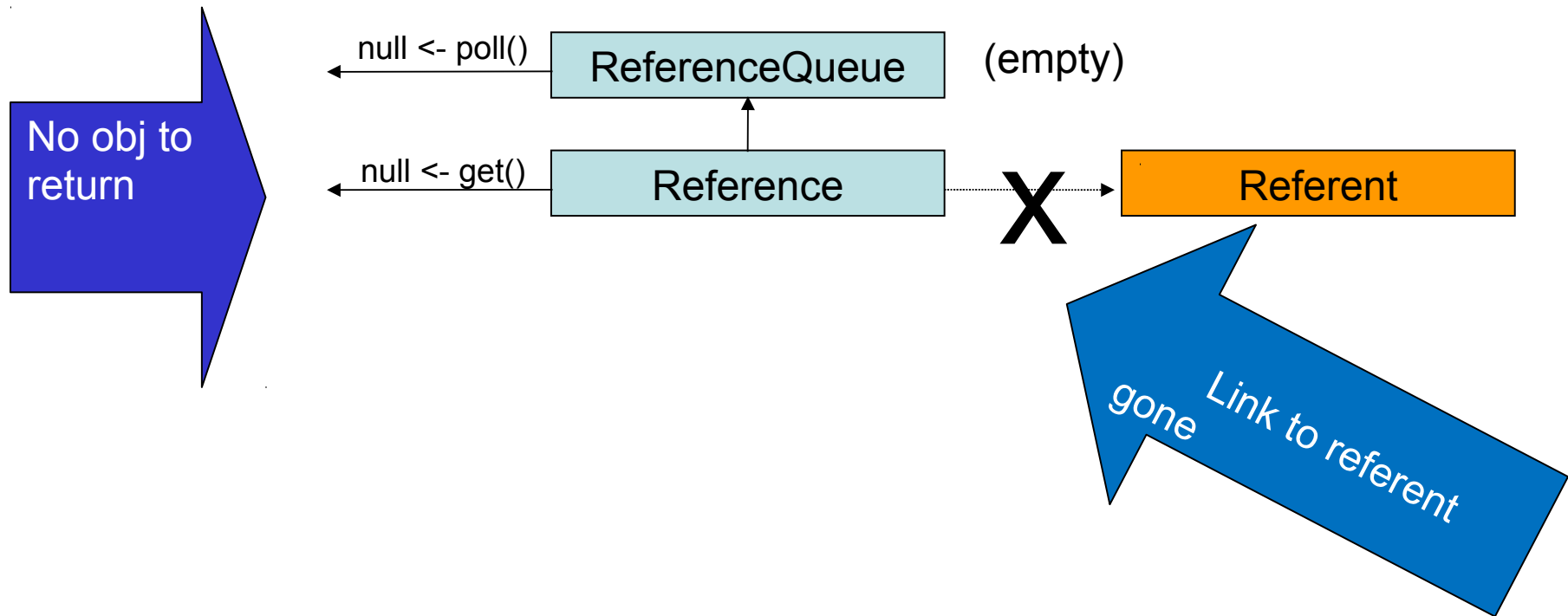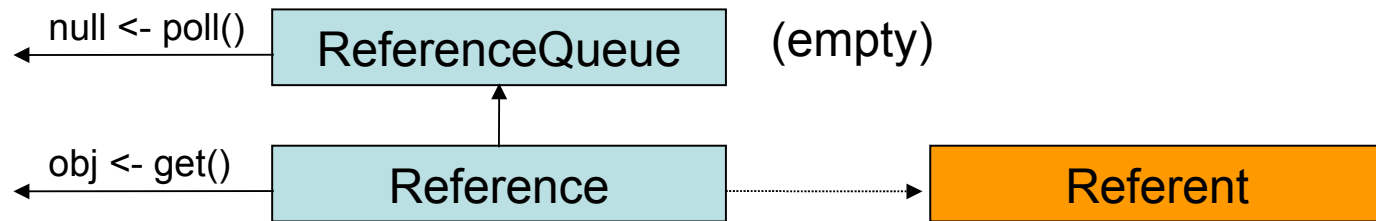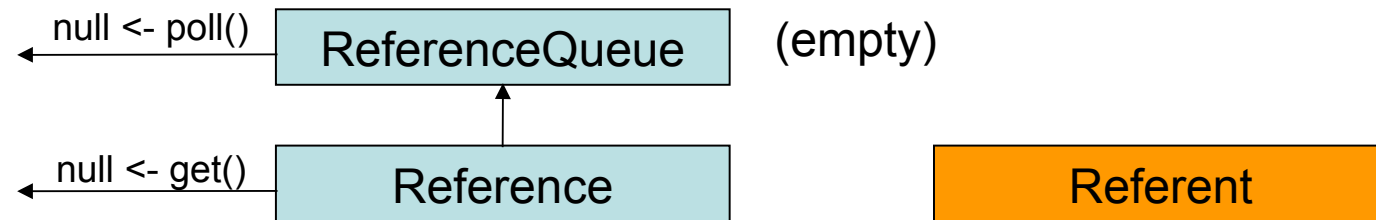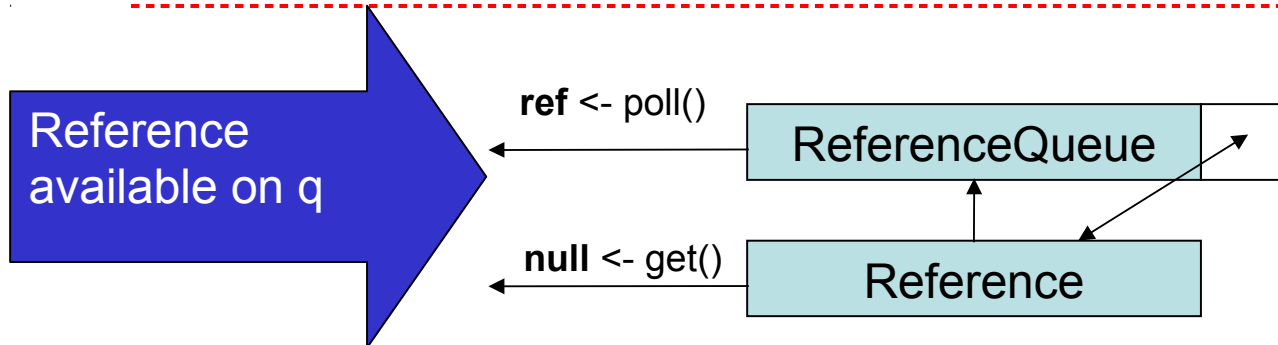
Java References

That looked easy…   So what's the catch?
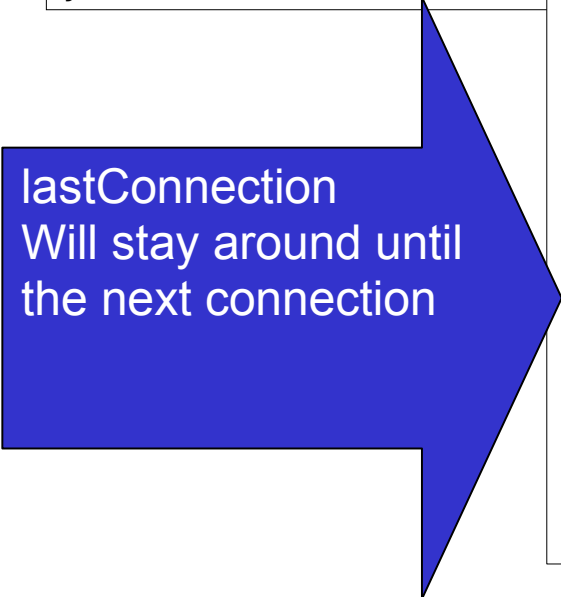
1. You mustn't have a strong reference to the **referent**
2. Your application has to handle processing the associated reference Q. Assuming you created one.
3. The **referent** is gone, gone, gone - any information you need to deal with "end of life" processing has to be held outside the object.
4. You must have a **strong** reference to the Reference if you want to be kept informed..
5. Java scope rules are not exactly how you might think.
6. The contract between your code and the Java Reference API is somewhat grey and foggy.

7,8,9… See rule 1

1. You mustn't have a strong reference to the **referent**

The whole point of using a Reference is **not** to have a strong reference
(at least in your code)

```
class MyConnectionReference extends WeakReference {
    Socket socket;
    public MyConnectionReference(Socket s,MyConnection
conn,ReferenceQueue q)        {
        super(conn,q);
        this.socket=s;
    }
}
```

```
…
private ServerSocket s=new ServerSocket();
private ReferenceQueue q=new ReferenceQueue();
private MyConnection lastConnection=null;
private List<MyConnectionReference> connections …


…

while(true) {
    Socket client=s.accept();
    lastConnection=buildConnection(client);
    connections.add(new MyConnectionReference(client, lastConnection));

}
```

lastConnection
Will stay around until
the next connection

2. Your application has to handle processing the associated reference queue. (assuming you created one).

If you did this :

```
ReferenceQueue q;
new Reference(foo,q);
```

Then you need to do this :

```
Reference next=q.poll();
if(next!=null)  handle(next);
```

Or:

```
while(true) {
    Reference next=q.remove();
    handle(next);
}
```

if you don't keep up with clearing the q: you may run out of memory
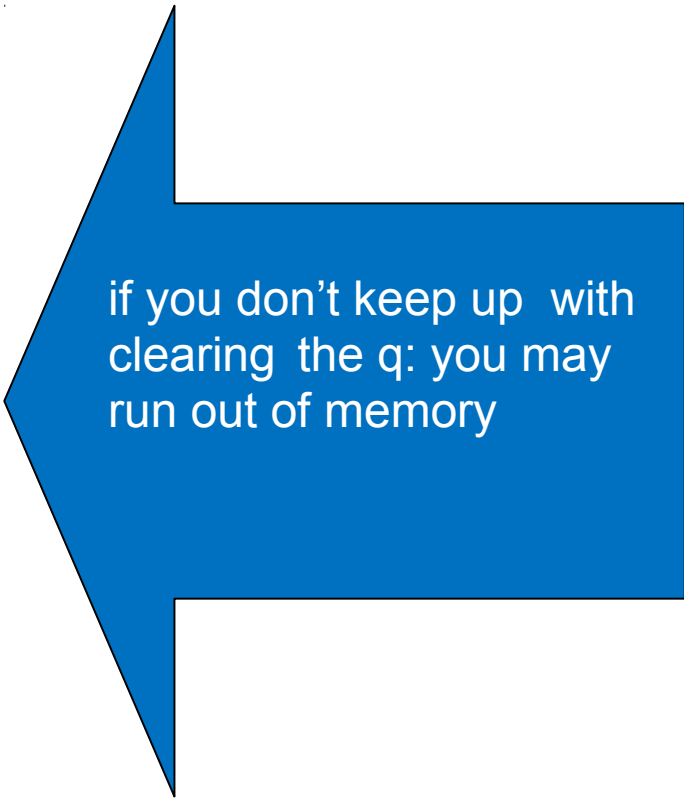
3 The **referent** is gone, gone, gone

I repeat: The **referent** is gone, gone, gone

any information you need to deal with "end of life" processing has to be held outside the object.

A simple solution is to subclass the Reference Object:

```
class MyConnectionReference extends WeakReference {
    Socket socket;
    public MyConnectionReference(Socket s,MyConnection conn,ReferenceQueue q)    {
        super(conn,q);
        this.socket=s;
    }
}
```
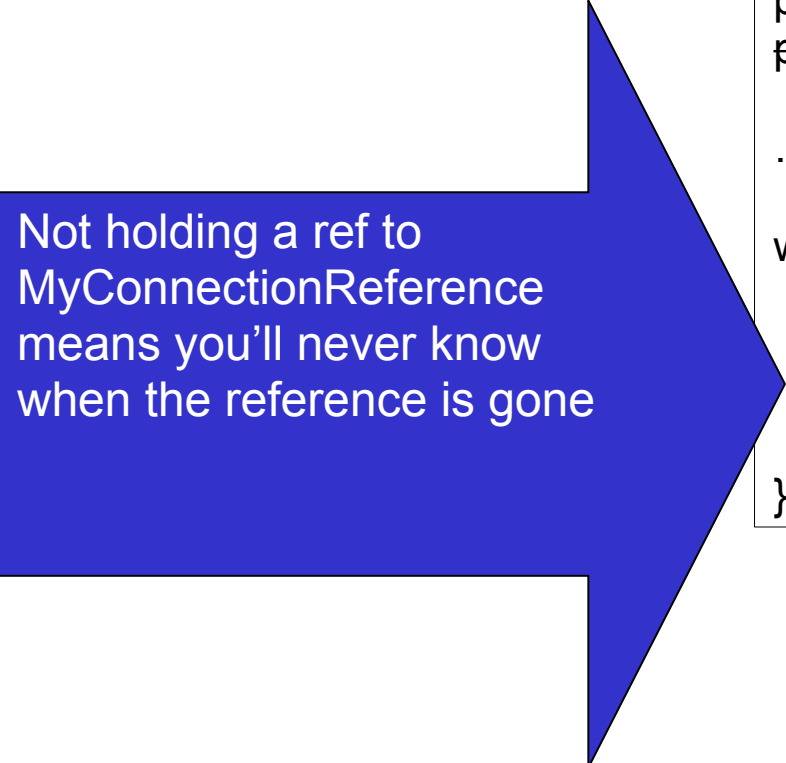
```
MyConnectionReference ref=queue.next();
ref.socket.close();
```

Simples
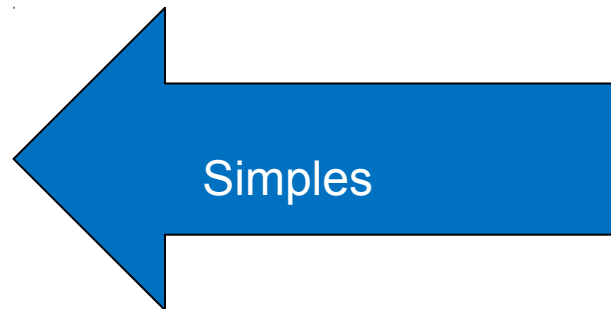
4. You must have a **strong** reference to the Reference if you want to be kept informed..

```
class MyConnectionReference extends WeakReference {
    Socket socket;
    public MyConnectionReference(Socket s,MyConnection
conn,ReferenceQueue q)      {
        super(conn,q);
        this.socket=s;
    }
}
```
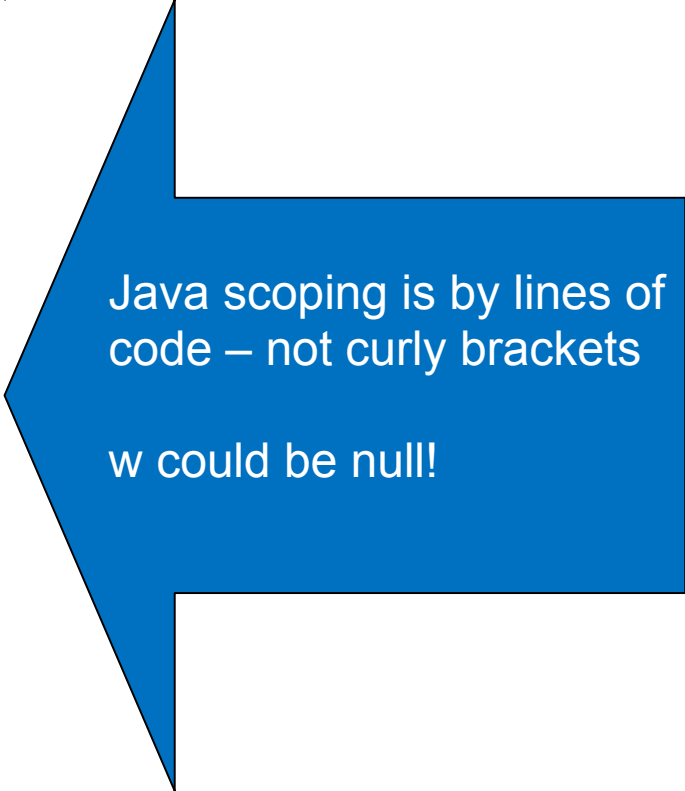
```
…
private ServerSocket s=new ServerSocket();
private ReferenceQueue q=new ReferenceQueue();
private MyConnection lastConnection=null;
private List<MyConnectionReference> connections …

…

while(true) {
    Socket client=s.accept();
    lastConnection=buildConnection(client);
    new MyConnectionReference(client, lastConnection);

}
```

Not holding a ref to MyConnectionReference means you'll never know when the reference is gone

5. Java scope rules are not exactly how you might think.

```
…
public void demo(String name) {

    String s="my name is "+name;

    WeakReference r=new WeakReference(s);

    String w=r.get();

    System.out.println(w);

}
```

Java scoping is by lines of code – not curly brackets

w could be null!

6. The contract between your code & the Java Reference API is somewhat grey and foggy.

The existence of a Garbage Collector is not part of the JVM specification or the the Java Language

The Java runtime does provides a connection to GC

There are rules – but exactly when you can expect your object to get collected, or when you can expect to be notified it has been collected – is generally implementation specific

When or if objects get GC'd , and hence drive Java Reference Queues - is determined by the GC implementation and various tuning parameters…

Don't rely on local implementation behaviour

Reference Object Types

Now we know a bit about the principle of Java Reference Object, why and how would we use them?

- Soft references:
  - Safety valve to avoid OutOfMemoryError

- WeakReference
  - Maintain data/association for an object without preventing garbage collection

- PhantomReference
  - Clean up resources after garbage collection

Soft Reference

- The **referent** of a Soft reference may be garbage collected once there are no remaining **strong** references to it in the application.

- However, the garbage collector will avoid garbage collecting a soft reference if possible (ie unless an out of memory condition would result)

- While alive, the **referent** may be retrieved via the Reference's get() method.

- If and when collected, get() will return null, and the Reference will be placed on the associated ReferenceQueue if any.

Soft Reference Use Cases

- The principal use cases for Soft References are:
  - Reclaimable caches
  - Safety valve for OutOfMemoryErrors

- These are closely related as one might expect

- Used for data which would be nice to keep, but nevertheless expendable, such as a cache or buffer.

- In the event that the JVM runs low on heap storage, the referent of the Soft Reference may be garbage collected, releasing some extra storage for the application to avoid out of memory.

- If used for a cache, it will then need to build up its contents again.

Weak References

- Weak References are extremely similar to Soft References in their use and operation.

- The only difference is that with a Weak Reference the garbage collector makes no attempt to delay collection of the referent – it's treated just like a "normal" object for garbage collection.

- Once its referent has been collected, the get() method of the WeakReference will return null, and is placed on the associated ReferenceQueue if one was specified on its constructor.

- Most often used in conjunction with a ReferenceQueue to notify collection of the WeakReference referent.

IBM

Weak Reference Use Cases

- In spite of the similarity with Soft References, the use cases for weak references are actually quite different because no attempt is made to delay garbage collection of a weakly reachable object.

- There are 2 main use cases:
  - Maintaining an association between objects which have no inherent relationship, while still allowing them to be garbage collected once they are no longer in active use.
  - Implementing a canonical map for a particular type of object.

- The real purpose behind weak references is to be able to hold references for as long as needed, without those references becoming the reason for holding on to them.

Phantom References

- Sole constructor PhantomReference(referent,queue) **requires** a ReferenceQueue
  - Though this may be **null**, that would be pointless

- PhantomReference get() method **always** returns **null**
  - Referent may not be retrieved by the application

- Physical heap storage associated with the referent is not collected until:
  - Referent is eligible for collection and finalized
  - PhantomReference is cleared or itself garbage collected

The API contract requires calling the clear() method on the Reference – even though the object has been garbage collected and cannot be accessed anyway

Phantom Reference Use Cases

- The principal use case for Phantom References is an alternative to finalizers for cleaning up the resources of objects which have been garbage collected.

- Some of the pitfalls of finalizers are avoided because the referent objects are not delayed in being garbage collected.

Summary

- Java References provide a degree of control over the behaviour of the garbage collector, and an alternative to finalize() for cleaning up resources when an object is collected.

- Soft References
  - Retained for as long as possible, but reclaimed in preference to out of memory error.
  - Used to release memory when needed instead of throwing OutOfMemoryError

- Weak References
  - Referents are reclaimed as normal by the Garbage Collector
  - Principally used to associate objects without causing them to be retained on heap
  - Used internally by WeakHashMap utility class.

- Phantom references
  - Referents cannot be retrieved via get(), reclaimed as normal
  - Provide an preferable alternative to finalize() for cleaning up collected objects
  - Always used with a ReferenceQueue

Wrapup

Java References are a better alternative to using finalizers - and have more usecases.

Using them for caching successfully depends on managing the size your cache. Unconstrained cache sizes can easily lead to poor performance profiles

Use WeakHashMap for simple, small caches etc.

For larger and more sophisticated usage – don't write your own!

Remember – in using References you are opting in to an event driven system that is primarily implementation specific. There are no real guarantees on compatible behaviour across JVM vendors, versions, or even configurations

References (no pun intended)

- IBM DeveloperWorks article by Brian Goetz on using Soft References:
  – http://www.ibm.com/developerworks/java/library/j-jtp01246/index.html

- IBM DeveloperWorks article by Brian Goetz on using the WeakHashMap:
  – http://www.ibm.com/developerworks/java/library/j-jtp11225/index.html

- A blog article on Java References by Keith D Gregory:
  – http://www.kdgregory.com/index.php?page=java.refobj

- Wikipedia article on weak references:
  – http://en.wikipedia.org/wiki/Weak_reference

- Javadoc for Package java.lang.ref (links to Reference and related classes)
  – http://download.oracle.com/javase/6/docs/api/java/lang/ref/package-summary.html

- Javadoc for WeakHashMap:
  – http://download.oracle.com/javase/6/docs/api/java/util/WeakHashMap.html

References

- **Get Products and Technologies:**
  - IBM Java Runtimes and SDKs:
    - https://www.ibm.com/developerworks/java/jdk/
  - IBM Monitoring and Diagnostic Tools for Java:
    - https://www.ibm.com/developerworks/java/jdk/tools/

- **Learn:**
  - IBM Java InfoCenter:
    - http://publib.boulder.ibm.com/infocenter/javasdk/v6r0/index.jsp

- **Discuss:**
  - IBM Java Runtimes and SDKs Forum:
    - http://www.ibm.com/developerworks/forums/forum.jspa?forumID=367&start=0

Copyright and Trademarks

© IBM Corporation 2011. All Rights Reserved.

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., and registered in many jurisdictions worldwide.

Other product and service names might be trademarks of IBM or other companies.

A current list of IBM trademarks is available on the Web – see the IBM "Copyright and trademark information" page at URL: www.ibm.com/legal/copytrade.shtml