Java Reference Objects

or

How I Learned to Stop Worrying and Love OutOfMemoryError

Contents

Object Life Cycle

Types of Reference Objects

Memory Management with Soft and Weak References

Replacing Finalizers by Phantom References

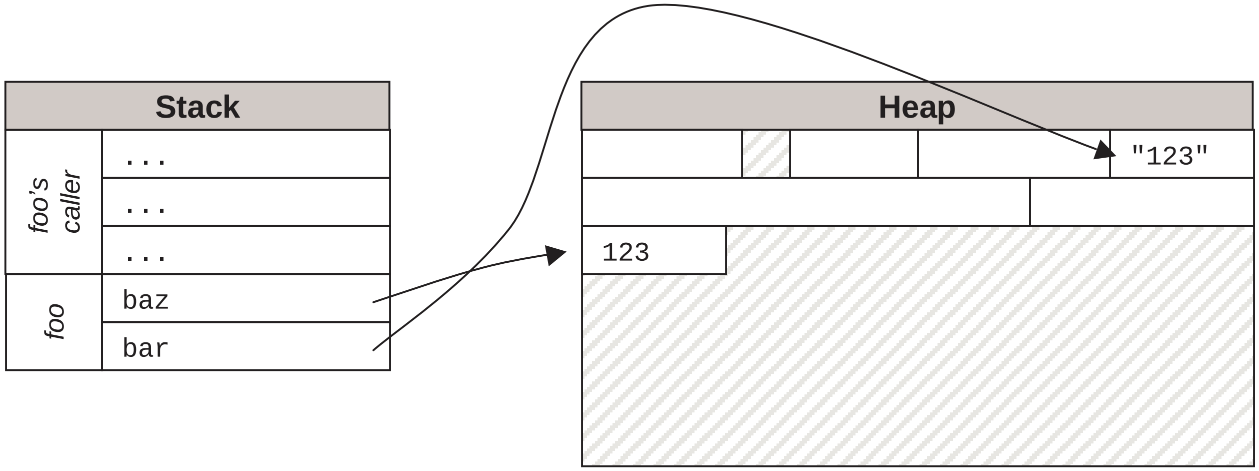
Unit Testing with Reference Objects

Role of Stack and Heap

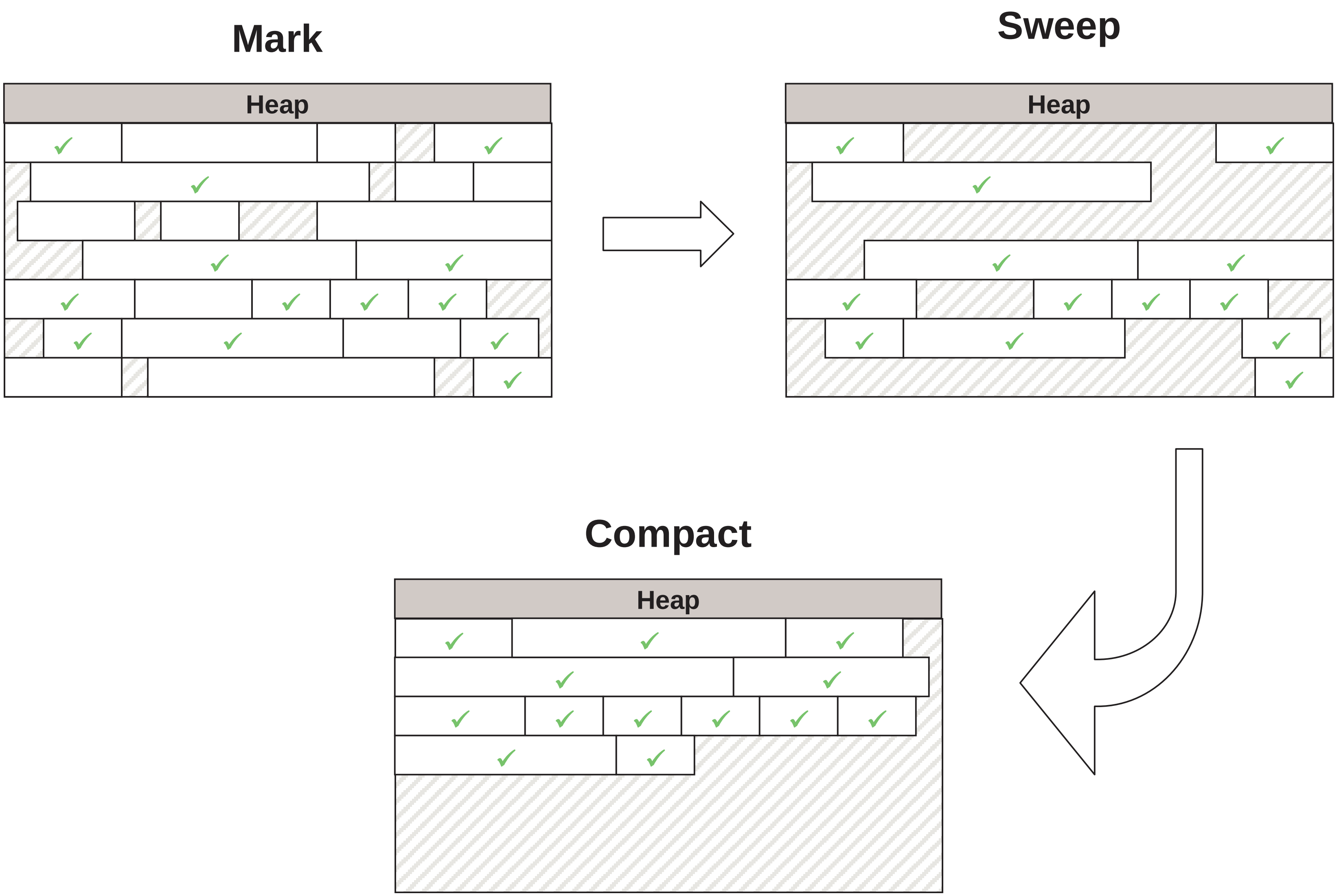
Stack holds all local variables, including method parameters and object references

Heap holds object data

```
public static void foo(String bar)
{
    Integer baz = new Integer(bar);
}
```



Garbage Collection Process



Mark

Heap

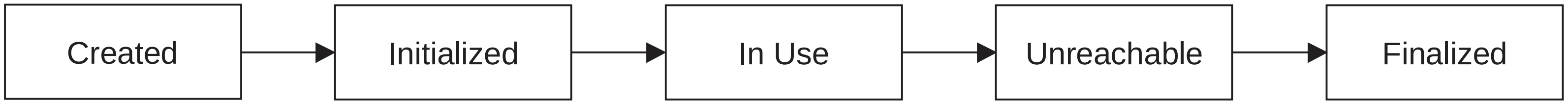
Sweep

Heap

Compact

Heap

Object Life Cycle pre Reference Objects


Created → Initialized → In Use → Unreachable → Finalized

`new` operator creates the object, constructor initializes it

- These are separate steps!

In-use (reachable) when program can access it

- Chain of references from static member variable, local method variable, or in-process expression

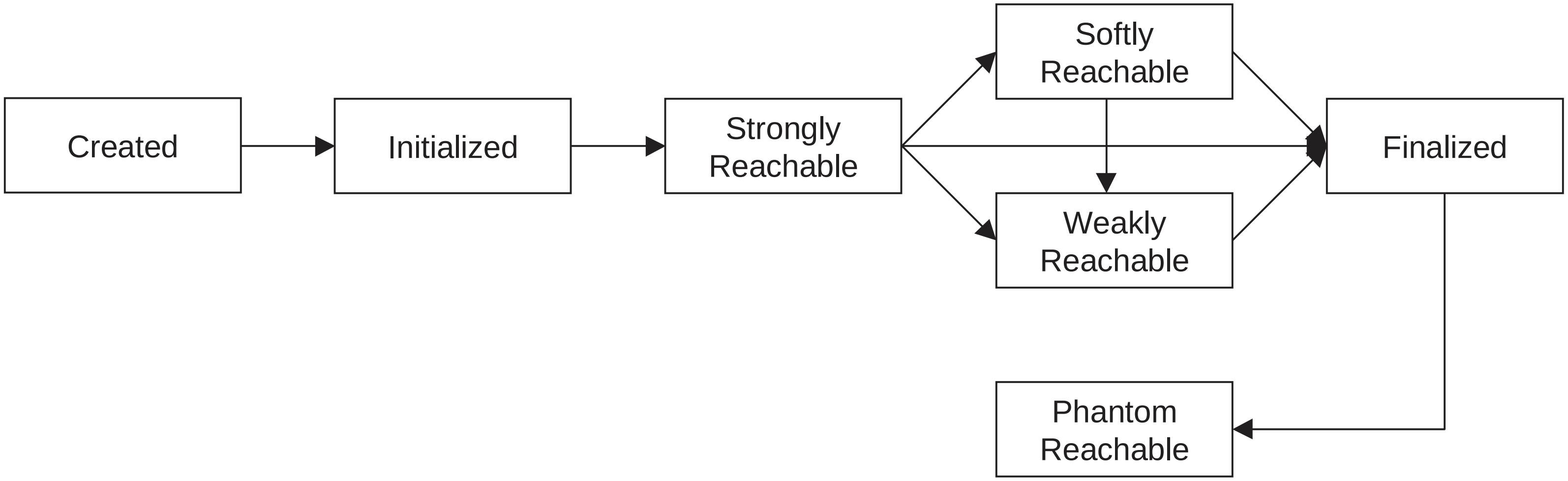
Unreachable when nothing points to it

- But the garbage collector only runs when JVM needs memory
- May never happen!

Finalizer is run after object is selected for collection

- Memory becomes available only after finalizer runs — if it exists

Object Life Cycle post Reference Objects

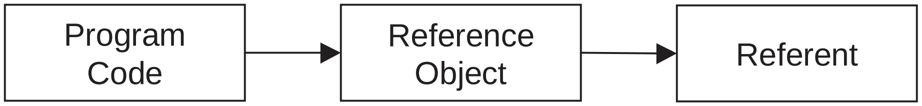


Unreachable objects are still eligible for collection

But there are different levels of unreachability

- Garbage collector is more/less aggressive
- Docs indicate strict hierarchy, that's misleading: reachability depends on the reference objects *you* use

How Reference Objects Work



Adds a layer of indirection

- Call `get()` on the reference object to access referent
- `get()` returns null when referent is collected (reference is “cleared”)

Program must hold a strong reference to the reference object itself

- Otherwise it will be collected

Program must hold strong reference to referent while accessing it

- Otherwise it might be reclaimed between two statements

Phantom references are … different

Types of Reference Objects

`SoftReference`

- Doesn’t prevent garbage collector from reclaiming referent, but asks nicely that it be left alone
- “Official” use: memory-sensitive cache
- Better use: circuit breaker

`WeakReference`

- Garbage collector will reclaim referent at the drop of a hat
- Useful when you want to attach data to an object with limited lifetime
- Or for a canonicalizing map

`PhantomReference`

- Lets program know when garbage collector has already marked referent for collection, allowing program-controlled cleanup
- Can’t be used to access referent directly — `get()` returns null

Reference Queues

Reference objects may be associated with a `ReferenceQueue` when created, will be added to that queue when cleared

- Program can poll `ReferenceQueue` to find cleared objects
- Must still hold a strong reference to the reference object, or it will be collected — queue doesn't hold strong reference

The only way to work with Phantom references

Also useful for cleaning up

- Can poll with a background thread
- Or just check the queue when creating new objects

Soft References as Circuit Breaker

Technique

- Hold large object via `SoftReference` while performing memory-intensive operations
- Switch to strong reference to update the large object
- If reference is cleared, operation fails

Rationale

- Memory consumption tends to be localized
- Failing single operation is better than throwing `OutOfMemoryError`

Not a silver bullet

- Always a window where `OutOfMemoryError` is possible
- Sometimes you can't control this (*eg*, DOM tree)

Code in need of a circuit breaker

```
public static List<List<Object>> processResults(ResultSet rslt)
throws SQLException
{
    try {
        List<List<Object>> results = new LinkedList<List<Object>>();
        ResultSetMetaData meta = rslt.getMetaData();
        int colCount = meta.getColumnCount();
        while (rslt.next())
        {
            List<Object> row = new ArrayList<Object>(colCount);
            for (int ii = 1 ; ii <= colCount ; ii++)
                row.add(rslt.getObject(ii));

            results.add(row);
        }
        return results;
    }
    finally {
        closeQuietly(rslt);
    }
}
```

Adding Soft References

```
SoftReference<List<List<Object>>> ref
    = new SoftReference<List<List<Object>>>(
        new LinkedList<List<Object>>());

while (rslt.next())
{
    List<Object> row = new ArrayList<Object>(colCount);
    for (int ii = 1 ; ii <= colCount ; ii++)
        row.add(rslt.getObject(ii));

    List<List<Object>> results = ref.get();
    if (results == null)
        throw new ResultsTooLargeException();
    else
        results.add(row);
    results = null;
}
```

Weak References for auto-clear cache

Often useful to attach data to an object via Map

- Particularly if you can't extend / decorate the original object
- However, a normal Map can turn into a memory leak, as it always holds a strong reference to the base object

If the map uses a weak reference, once the program is done with the object the associated data goes as well

- Example: `ThreadLocal`
- Should be used by `ObjectOutputStream`, but isn't

JDK provides `WeakHashMap`

- Keys are held by weak references, values by strong references
- When the weak references are cleared, map entry is removed

Canonicalizing Maps

Always returns the same value for a given key

- Think `String.intern()`

Useful when processing data with duplicates

- Pass raw data through map, replace duplicated objects with canonical object
- If there isn't a strong reference to the object, no need to hold it in the map — replace it next time through

Both key and value must be held via weak reference

- `WeakHashMap` isn't sufficient on its own
- But it provides a good starting point

Interning strings via Weak References

```
private Map<String,WeakReference<String>> _map
        = new WeakHashMap<String,WeakReference<String>>();


public static synchronized String intern(String str)
{
    WeakReference<String> ref = _map.get(str);
    String s2 = (ref != null) ? ref.get() : null;
    if (s2 != null)
        return s2;
    _map.put(str, new WeakReference(str));
    return str;
}
```

The Trouble with Finalizers

Finalizers introduce a break between identifying a dead object and reclaiming its memory

- Dead objects go into finalization queue
- If every dead object has a finalizer, you'll get OOM

Finalization takes place on a separate thread

- In practice, just one thread
- A slow finalizer can leave the heap full of uncollected objects

Finalizer may never run

- Only run when when GC identifies object as dead — if GC doesn't run, finalizer isn't executed
- This applies to phantom references as well, but your program can iterate over the references manually

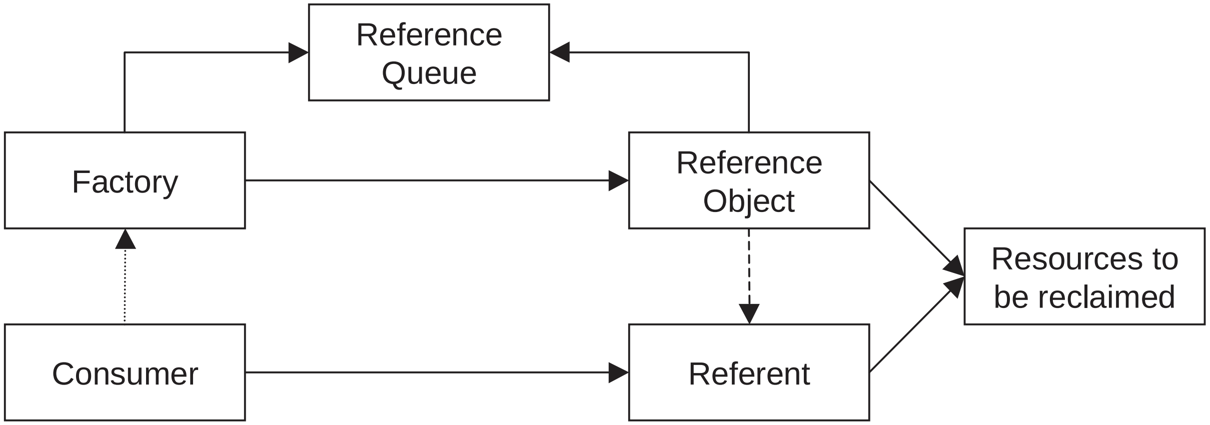
Using Phantom References

The phantom reference must be associated with a `ReferenceQueue`

- The reference is enqueued when its referent is marked for collection
- The memory is not freed until the reference is dequeued!

Program accesses the referent directly, lets it go out of scope

- Must keep a separate (strong) reference to the resources



Phantom Reference Example

Database connection pool

- Wraps actual connection, returns wrapper
- Connection returns to pool via `close()` or wrapper collection

```
private ReferenceQueue<Object> _refQueue = // ...
private IdentityHashMap<Object,Connection> _ref2Cxt = // ...
private IdentityHashMap<Connection,Object> _cxt2Ref = // ...

// ...

private Connection wrapConnection(Connection cxt)
{
    Connection wrapped = new PooledConnection(this, cxt);
    PhantomReference<Connection> ref =
            new PhantomReference<Connection>(wrapped, _refQueue);
    _cxt2Ref.put(wrapped, ref);
    _ref2Cxt.put(ref, wrapped);
    return wrapped;
}
```

Unit Testing and Reference Objects

It isn’t easy

- Running out of memory is harder than it looks
- System.gc() is just a hint
- Make sure that you don’t hold strong references to the referent

But you have to do it

- Reference objects become useful when living on the edge — too easy to fall off if you don’t test

Build task-specific scaffolding

- Example: `ResultSet` implementation that returns large `byte[]`s on every call to `getObject()`

Write development-only tests

- Sometimes Hotspot gets in the way

Additional Reading

The “companion volume” to this presentation.

- `http://www.kdgregory.com/index.php?page=java.refobj`

Sun’s current whitepaper on tuning the garbage collector, which provides some good background information on how the collector works (Sun JVM only).

- `http://java.sun.com/docs/hotspot/gc5.0/gc_tuning_5.html`

An article from Brian Goetz, about using Weak references to associate objects with limited lifetimes. I don’t often use this technique, so only touched on it lightly in this presentation. I recommend reading his entire series of articles.

- `http://www.ibm.com/developerworks/java/library/j-jtp11225/index.html?S_TACT=105AGX02&S_CMP=EDU`