In a language without garbage collection, such as C++, significant attention must be paid to memory management. For each dynamic object, you must either implement reference counting to simulate the effect of garbage collection, or manage the "ownership" of each object -- identifying which class is responsible for deleting an object. Such ownership is usually not maintained declaratively, but rather by (often undocumented) convention. While garbage collection means that Java developers don't have to worry (much) about memory leaks, sometimes we still do have to worry about object ownership to prevent data races and unwanted side effects. In this article, Brian Goetz identifies some of the situations where Java developers must pay attention to object ownership.

If you learned to program before 1997, chances are that the first programming language you learned didn't provide transparent garbage collection. Each new operation had to be balanced by a corresponding delete, or else your program would leak memory, and eventually the memory allocator would fail and your program would crash. Whenever you allocated an object with new, you had to ask yourself, who will delete this object and when?

Aliases, also known as ...

A primary contributor to the complexity of memory management is aliasing: having more than one copy of a pointer or reference to the same block of memory or object. Aliases occur naturally all the time. For example, in Listing 1 there are at least four references to the something object created on the first line of makeSomething:

- The something reference
- At least one reference held within collection c1
- The temporary aSomething reference created when something is passed as an argument to registerSomething
• At least one reference held within collection c2

### Listing 1. Aliases in typical code

```java
collection c1, c2;

public void makeSomething {
    something = new Something();
    c1.add(something);
    registerSomething(something);
}

private void registerSomething(Something aSomething) {
    c2.add(aSomething);
}
```

There are two major memory-management hazards to avoid in non-garbage-collected languages: memory leaks and dangling pointers. To prevent memory leaks, you must ensure that each allocated object is eventually deleted. To avoid dangling pointers (the dangerous situation where a block of memory is freed but a pointer still references it), you must delete the object only after the last reference is released. The mental and digital bookkeeping required to satisfy these two constraints can be significant.

### Managing object ownership for memory management

Besides garbage collection, two other approaches are commonly used to deal with the problems of aliasing: reference counting and ownership management. **Reference counting** involves keeping a count of how many references to a given object are extant, then deleting the object automatically after the last one is released. In C -- and in most versions of C++ prior to the mid-1990s -- this was impossible to do automatically. The Standard Template Library (STL) allows for the creation of "smart" pointers that can automatically do reference counting (see the `shared_ptr` class from the open source Boost library, or the simpler `auto_ptr` class from the STL for examples).

**Ownership management** is the process of designating one pointer to be the "owning" pointer and all other aliases to be only temporary second-class copies, and deleting the object only when the owning pointer is released. In some cases, ownership can be "transferred" from one pointer to another, such as a method for writing to a socket that accepts a buffer as an argument and that deletes the buffer when the write completes (such methods are often called *sinks*). In this case, the ownership of the buffer has been effectively transferred, and the calling code must assume the buffer to have been deleted when the transferee method returns. (Ownership management can be further simplified by ensuring that all alias pointers have scope that is tied to the call stack, such as method parameters or local variables, and by copying the object if a reference is going to be held by a non-stack-scoped variable.)

### So what?

At this point, you're probably wondering why I'm even talking about memory management, aliases, and object ownership. After all, garbage collection is one of the core features of the Java language, and memory management is a chore of the past! Just let the garbage collector handle it; that's its job. Those who have been freed from the chore of memory management don't want to go
back, and those who never had to deal with it can't even imagine how horrible life must have been as a programmer in the bad old days -- like 1996.

**Watch out for dangling aliases**

So does that mean we can say goodbye to the concept of object ownership? Yes ... and no. Garbage collection does indeed obviate the need for explicit resource deallocation, most of the time. (I'll discuss some exceptions in a future column.) However, there is one area where ownership management is still very much an issue in Java programs, and that is the problem of **dangling aliases**. Java developers often rely on implicit assumptions of object ownership to determine which references should be considered read-only (a `const` pointer, in C++ parlance) and which can be used to modify the referenced object's state. A dangling alias occurs when two classes each think (mistakenly) that they hold the only writeable reference to a given object. When this happens, one or both of the classes will be confused when the object's state changes unexpectedly.

**Case in Point**

Consider the code in Listing 2, where a UI component holds a `Point` object to represent its location. When `MathUtil.calculateDistance` is called to calculate how far the object has moved, we are relying on an implicit and subtle assumption -- that `calculateDistance` will not mutate the state of the `Point` objects passed to it, or worse, maintain a reference to the `Point` objects (such as by storing them in a collection or passing them to another thread) that might then later be used to mutate their state after `calculateDistance` returns. In the case of `calculateDistance`, it might seem absurd to worry about such behavior, as this would clearly be an unspeakable breach of etiquette. But by passing a mutable object to another method, it is simply an act of faith that the object will be returned to you unharmed and that there will be no future undocumented side effects on the object's state (such as the method sharing the reference with another thread, which might wait five minutes and then change the object's state).

**Listing 2. Passing mutable objects to foreign methods is an act of faith**

```java
private Point initialLocation, currentLocation;

public Widget(Point initialLocation) {
    this.initialLocation = initialLocation;
    this.currentLocation = initialLocation;
}

public double getDistanceMoved() {
    return MathUtil.calculateDistance(initialLocation, currentLocation);
}

// The ill-behaved utility class MathUtil
public static double calculateDistance(Point p1, Point p2) {
    double distance = Math.sqrt((p2.x - p1.x) ^ 2 + (p2.y - p1.y) ^ 2);
    p2.x = p1.x;
    p2.y = p1.y;
    return distance;
}
```
What a silly example

The obvious and universal response to this example -- namely, that it is a silly example -- merely underscores the fact that the concept of object ownership is alive and well, and merely undocumented, in Java programs. The calculateDistance method shouldn't mutate the state of its arguments because it doesn't "own" them -- the calling method does, of course. So much for not having to think about object ownership.

Here's a more practical example of the confusion that can be caused by not knowing who owns an object. Consider again a UI component that has a Point property to represent its location. Listing 3 shows three ways to implement the accessor methods setLocation and getLocation. The first way is the laziest and offers the highest performance, but it has several vulnerabilities to both deliberate attacks and innocent mistakes.

**Listing 3. Value and reference semantics for getters and setters**

```java
public class Widget {
    private Point location;

    // Version 1: No copying -- getter and setter implement reference semantics
    // This approach effectively assumes that we are transferring ownership of the Point from the caller to the Widget, but this assumption is rarely explicitly documented.
    public void setLocation(Point p) {
        this.location = p;
    }

    public Point getLocation() {
        return location;
    }

    // Version 2: Defensive copy on setter, implementing value semantics for the setter
    // This approach effectively assumes that callers of getLocation will respect the assumption that the Widget owns the Point, but this assumption is rarely documented.
    public void setLocation(Point p) {
        this.location = new Point(p.x, p.y);
    }

    public Point getLocation() {
        return location;
    }

    // Version 3: Defensive copy on getter and setter, implementing true value semantics, at a performance cost
    public void setLocation(Point p) {
        this.location = new Point(p.x, p.y);
    }

    public Point getLocation() {
        return (Point) location.clone();
    }
}
```

Now, consider this innocent-looking use of setLocation:
Widget w1, w2;
  ...
  Point p = new Point();
  p.x = p.y = 1;
  w1.setLocation(p);
  p.x = p.y = 2;
  w2.setLocation(p);

Or this:

w2.setLocation(w1.getLocation());

Under version 1 of the setLocation/getLocation accessor implementation, it may look like the location of the first widget will be (1, 1) and the second will be (2, 2), but, in fact, both will be (2, 2). This may well be confusing both to the caller (because the first widget got moved unexpectedly) and to the widget class (because its location changed without the widget code being involved). In the second example, you may think you are moving widget w2 to where widget w1 is currently located, but you're actually constraining w2 to follow w1 every time w1 moves.

**Defensive copies**

Version 2 of setLocation does better: it makes a copy of the argument passed to it to ensure that there are no aliases to the Point that could mutate its state unexpectedly. But it doesn't go far enough either, because the following code will also have the (likely undesired) effect of moving the Widget without its knowledge:

Point p = w1.getLocation();
  ...
  p.x = 0;

Version 3 of getLocation and setLocation are fully safe against malicious or careless uses of alias references. This safety comes at some performance cost: the creation of a new object every time a getter or setter is called.

The different versions of getLocation and setLocation have different semantics, often referred to as value semantics (version 3) and reference semantics (version 1). Unfortunately, which semantics the implementer intends is rarely documented. The result is that users of the class don't know, and therefore should assume the worst.

The technique used by version 3 of getLocation and setLocation is called defensive copying, and despite the obvious performance cost, you should get in the habit of using it nearly all the time when returning or storing references to mutable objects or arrays, especially if you are writing a general-purpose facility that may be called by code that you haven't personally written (and often even then). Cases where an aliased mutable object gets unexpectedly modified can crop up in a number of subtle and surprising ways, and debugging them can be extremely difficult.

But it gets worse. Let's say that as a user of the Widget class, you don't know whether the accessors have value or reference semantics. Prudence would dictate that defensive copies are
needed when calling them, too. So, if you wanted to move w2 to the current location of w1, you would have to do this:

```java
Point p = w1.getLocation();
w2.setLocation(new Point(p.x, p.y));
```

If Widget implemented its accessors as in version 2 or 3, now we'd be creating two temporary objects for every call -- one outside the call to `setLocation` and one inside.

**Document accessor semantics**

The real problem with version 1 of `getLocation` and `setLocation` is not that they are vulnerable to confusing alias side effects (which they are), but that their semantics were not properly documented. If accessors were clearly documented to have reference semantics (instead of the commonly assumed value semantics), callers might be more likely to realize that when they call `setLocation`, they are transferring the ownership of that `Point` object to another entity, and be less likely to assume they still own it and can reuse it.

**Immutability to the rescue**

These problems with `Point` could have easily been solved if `Point` had been made immutable in the first place. There can be no side effects on immutable objects, and caching a reference to an immutable object is always safe from alias problems. All of the questions about the semantics of the `setLocation` and `getLocation` accessors are unambiguously determined if `Point` is immutable. Accessors for immutable properties will always have value semantics and do not need the defensive copying on either side of the call, making them more efficient.

So why wasn't `Point` made immutable in the first place? It was probably for performance reasons; early JVMs had less efficient garbage collectors. The object creation overhead of creating a new `Point` every time an object on the screen moved (even the mouse) probably seemed daunting at the time, and the overhead of making defensive copies probably seemed out of the question.

In hindsight, the decision to make `Point` mutable turned out to be costly to program clarity and performance. The mutability of the `Point` class creates a documentation burden for every method that accepts a `Point` parameter or returns a `Point`. Namely, does it mutate the `Point` or retain a reference to it after it returns? Given that so few classes actually include such documentation, the safe strategy when calling a method that doesn't document its call semantics or side-effect behavior is to create a defensive copy before passing it to any such method.

Ironically, the performance benefit of the decision to make `Point` mutable is dwarfed by the additional cost of the defensive copying required by `Point`’s mutability. In the absence of clear documentation (or a leap of faith), defensive copying is required on both sides of a method call -- by the caller, because it doesn't know if the callee will rudely mutate the `Point`, and by the callee, if it is retaining a reference to the `Point`. 
A real-world example

Here is another example of a dangling alias problem, which is very similar to one I recently saw in a server application. This application used publish-and-subscribe messaging internally to communicate events and status updates to other agents within the server. Agents could subscribe to whichever message streams interested them. Once published, messages delivered to other agents would probably be processed at a later time in a different thread.

Listing 4 shows a typical messaging event, posting notification of a new high bid in an auction system, and the code that generates it. Unfortunately, the interaction of the messaging event implementation and the caller implementation conspire to create a dangling alias. By simply copying the array reference instead of cloning it, both the message and the class that produces it hold a reference to the master copy of the previous bids array. If there is any delay between the time the message is published and the time it is consumed, subscribers could see different values for the previous5Bids array than was current at the time of publication, and multiple subscribers might see different values for the previous bids from each other. In this case, the subscriber would see a historical value of the current bid, and more up-to-date values of the previous bids, creating the illusion that the previous bids were higher than the current bid. It's not hard to imagine how this would cause problems -- and worse, such a problem would only manifest itself when the application is under heavy load. Making the message classes immutable and cloning mutable references such as arrays at construction time would have prevented this problem.

Listing 4. Dangling array alias in publish-subscribe messaging code

```java
public interface MessagingEvent {
    ...}

public class CurrentBidEvent implements MessagingEvent {
    public final int currentBid;
    public final int[] previous5Bids;

    public CurrentBidEvent(int currentBid, int[] previousBids) {
        this.currentBid = currentBid;
        // Danger -- copying array reference instead of values
        this.previous5Bids = previous5Bids;
    }
    
    // Now, somewhere in the bid-processing code, we create a
    // CurrentBidEvent and publish it.
    public void newBid(int newBid) {
        if (newBid > currentBid) {
            for (int i=1; i<5; i++)
                previous5Bids[i] = previous5Bids[i-1];
            previous5Bids[0] = currentBid;
            currentBid = newBid;

            messagingTopic.publish(new CurrentBidEvent(currentBid, previousBids));
        }
    }
}
```
Guidelines for mutable objects

If you are creating a mutable class $M$, you should be prepared to write a lot more documentation about the treatment of references to $M$ than if $M$ were immutable. First, you must choose whether methods that accept $M$ as arguments or return an $M$ object will use value or reference semantics, and be prepared to document that clearly in every other class that uses $M$ in its interface. If any methods that accept or return an $M$ object are implicitly assuming that the ownership of $M$ is being transferred, you must document that, too. You must also be prepared to accept the performance overhead of making defensive copies where necessary.

One special case where we are forced to deal with the issue of object ownership is with arrays, as arrays cannot be immutable. There may be a cost to making defensive copies when passing an array reference to another class, but unless you are sure that the other class either makes its own copy or that it will not hold the reference for longer than the duration of the call, you probably want to make a copy before passing the array. Otherwise, you can easily end up in a situation where the classes on both sides of the call implicitly assume that they own that array, with unpredictable results.

Summary

Dealing with mutable classes requires a lot more care than immutable classes. When passing references to mutable objects between methods, you need to clearly document under what cases the object's ownership is being transferred. And in the absence of clear documentation, you must make defensive copies on both sides of a method call. While the justification for mutability is often based on performance, because there is no need to create a new object every time its state changes, the performance cost of defensive copying can easily outweigh the performance savings from reduced object creation.
Related topics

- Read the complete *Java theory and practice* series by Brian Goetz. The following columns are particularly relevant to this one:
  - "To mutate or not to mutate?" (*developerWorks*, February 2003) explores the many benefits of immutability.
  - "Urban performance legends" (*developerWorks*, April 2003) further explores the myth that mutable objects are better for performance.
- Praxis 63 of Peter Haggar's *Practical Java Programming Language Guide* also encourages the use of immutable classes.
- Using auto_ptr Effectively from the *C++ User's Journal* explores the issue of object ownership, sources, and sinks in C++ programs.
- The Boost library is an open source library of useful C++ classes. The section on Smart Pointers illustrates how reference counting for memory management can be achieved -- mostly transparently -- in C++.