ObjC Memory Management and JNI

This page assumes you are already familiar with the following concepts:

- Allocating objects

There are a few simple rules to follow when writing JNI code that handles Cocoa objects:

Always use JNF_COCOA_ENTER()/JNF_COCOA_EXIT()

- These macros ensure that an autorelease pool is always setup and popped so ObjC objects are not leaked in RR mode.
- They also catch ObjC exceptions that are thrown, and rethrow them as Java exceptions

As a rule, Java objects should own native objects in jlongs

- The jlong type is used because it is big enough to hold both 32 and 64-bit sized pointers
    - Use the jlong_to_ptr() and ptr_to_jlong() macros to correctly handle casting and avoid sign-extension problems
- This keeps the number of JNI global refs HotSpot has to manage to a minimum
- Java objects that own native objects have to concretely define the lifecycle of the native objects they hold
    - Do not let native objects hold onto Java objects in JNI global refs, or you can create a cycle across the Java and ObjC reference count, and neither system will realize that they can deallocate their respective objects

Native objects held by Java objects must have a "hard" CF-retain count of 1 before being passed up to Java

- ObjC objects that are +alloc'd or -retained are not actually pinned in GC-mode unless they have been CFRetain()'d
    - As a counterpoint, any CFRetain()'d ObjC object must be -released or -autoreleased for it's retain count to remain balanced in RR mode
- When the Java object is done with the native object, it must be explicitly "hard" CFRelease()'d
    - This balances the "hard" CFRetain() which occurred before it was passed up to Java
- You need to determine if the object is safe to CFRelease() from any thread, or must only be done from the main AppKit thread
    - In some cases, releasing an object may do nothing but decrease it's retain count, but in other cases it will cause the object's -dealloc or -finalize method to be called, and those may not be any-thread safe if they end up releasing AppKit objects

Don't manage JNI global refs yourself

- If you need to pass a Java object through native (like to the main AppKit thread, or inside of an NSArray), use JNFJObjectWrapper or JNFJObjectWeakWrapper
    - It handles all the details of managing a JNI global ref in a native ObjC object
- If you have a JNIEnv when you are done with the JNFJObjectWrapper, pre-clear the reference
    - It is more efficient than the wrapper's -dealloc, which may have to connect the current thread to the JVM (possibly a ObjC-GC collector thread), just for the purposes of deleting the ref.

Don't put ObjC objects into C-structs

- The ObjC garbage collector can't find objects that have been stuffed into C arrays and structs, and it's an assignment error in ARC
- If you **must** use a struct to hold onto a ObjC object (like some pieces of the graphics code currently does), you need to pin the ObjC using CFRetain(), just like if you were going to push it up into the JVM heap.
    - Be sure to CFRelease() any AppKit objects on the main AppKit thread, so their -dealloc methods will be called on the right thread!