NSAutoreleasePool

Inherits From: NSObject

Declared In: foundation/NSAutoreleasePool.h

Class Description

The Foundation Kit uses the NSAutoreleasePool class to implement NSObject's **autorelease** method. An autorelease pool simply contains other objects, and when deallocated sends a **release** message to each of those objects. An object can be put into the same pool several times, and receives a **release** message for each time it was put into the pool. This class specification presents information on fine-tuning your application's handling of autorelease pools; see "Object Ownership and Automatic Disposal" in the introduction to the Foundation Kit for basic information on using the autorelease feature.

You use autorelease pools to limit the time an object remains valid after it's been autoreleased (that is, after it's been sent an **autorelease** message or has otherwise been added to an autorelease pool). You create an autorelease pool with the usual **alloc** and **init** messages, and dispose of it with **release**. An autorelease pool should always be released in the same context (invocation of a method or function, or body of a loop) that it was created. You should never send **retain** or **autorelease** to an autorelease pool.

Autorelease pools are automatically created and destroyed in NEXTSTEP applications, so your code normally doesn't have to worry about them. There are two cases, though, where you should explicitly create and destroy your own autorelease pools. If you're writing a program that's not based on the Application Kit, such as a UNIX tool, there's no built-in support for autorelease pools; you must create and destroy them yourself. Also, if you need to write a loop that creates many temporary objects, you should create an autorelease pool in the loop to prevent too long a delay in the disposal of those objects.

Enabling the autorelease feature in a program that's not based on the Application Kit is fairly easy. Many programs have a top-level loop where they do most of their work. To enable the autorelease feature you create an autorelease pool at the beginning of this loop and release it at the end. An **autorelease** message sent in the body of the loop automatically puts its receiver into this pool. Your **main()** function might look like this:

```
int main(int argc, char *argv[])
{
    int i;

 /* Do whatever setup is needed. */

    for (i = 0; i < argc; i++) {
        NSAutoreleasePool *pool;
        NSString *fileContents;
        NSString *fileName;

        pool = [[NSAutoreleasePool alloc] init];
        fileName = [NSString stringWithCString:argv[i]];

        fileContents = [[[NSString alloc]
            initWithContentsOfFile:fileName] autorelease];

        processFile(fileContents);

        [pool release];
    }

    /* Do whatever cleanup is needed. */

    exit(EXIT_SUCCESS);
}
```

Any object autoreleased inside the **for** loop, such as the **fileContents** string object, is added to **pool**, and when **pool** is released at the end of the loop those objects are also released.

Note that autoreleasing doesn't work outside of the loop. This isn't a problem, since the program terminates shortly after the loop ends, and memory leaks aren't usually serious at that stage of execution. Your cleanup code shouldn't refer to any objects created inside the loop, though, since they may be autoreleased in the loop and therefore released as soon as it ends.

Nesting Autorelease Pools

You may need to manually create and destroy autorelease pools even in a NEXTSTEP application if you write loops that create many temporary objects. For example, if you write a loop that iterates 1000 times and invokes a method that creates 15 temporary objects, those 15,000 objects will remain until the application's autorelease pool is deallocated, well after they're no longer needed.

You can create your own autorelease pools within the loop to prevent these unwanted objects from remaining around. Autorelease pools nest themselves on a per-thread basis,

so that if you create your own pool, it adds itself to the application's default pool, forming a stack of autorelease pools. Likewise, if you create another pool (within a nested loop, perhaps), it adds itself to the first pool you created. **autorelease** automatically adds its receiver to the last pool created, creating a nesting of autorelease contexts. The implications of this are described below.

A method that creates autorelease pools looks much like the **main()** function given above:

```
- (void)processString:(NSString *)aString
{
    int i;

    for (i = 0; i < 1000; i++) {
        NSAutoreleasePool *subpool = [[NSAutoreleasePool alloc] init];
        NSString *thisLine;

        thisLine = [self getLineNumbered:i fromString:aString];
        /* Do some work with thisLine. */

        [subpool release];
    }

    return;
}
```

If you assume that **getLineNumbered:fromString:** returns a string object that's been autoreleased while **subpool** is in effect, that object is released with **subpool** at the end of the loop. The work involving **thisLine** may create other temporary objects, which are also released at the end of the loop. None of these objects remains outside of this loop or the **processString:** method (unless they've been retained).

Note that because an autorelease pool adds itself to the previous pool when created, it doesn't cause a memory leak in the face of an exception or other sudden transfer out of the current context. If an exception occurs in the above loop, or if the work in the loop involves immediately returning or breaking out of the loop, the sub-pool is released by the application's default pool (or whatever pool was in effect before the sub-pool was created), "unwinding" the autorelease-pool stack up to the one that's supposed to be active.

Guaranteeing the Foundation Ownership Policy

By manually creating an autorelease pool, you reduce the potential lifetime of temporary objects to the lifetime of that pool. After an autorelease pool is deallocated, you should regard as "disposed of" any object that was autoreleased while that pool was effective, and not send a message to that object or return it to the invoker of your method. This method, for example, is incorrect:

```
- findMatchingObject:anObject
{
    id match;

    match = nil;
    while (match == nil) {
        NSAutoreleasePool *subpool = [[NSAutoreleasePool alloc] init];

       /*
        * Do some searching that creates a lot of temporary objects.
        */
        match = [self expensiveSearchForObject:anObject];

        [subpool release];
    }

    [match setIsMatch:YES forObject:anObject];

    return match;
}
```

expensiveSearchForObject: is invoked while **subpool** is in effect, which means that **match**, which may have been autoreleased, is released at the bottom of the loop. Sending **setIsMatch:forObject:** after the loop could cause the application to crash. Similarly, returning **match** allows the sender of **findMatchingObject:** to send a message to it, also causing your application to crash.

If you must pull a temporary object out of a nested autorelease context, you can do so by retaining the object within the context and then autoreleasing it after the pool has been released. Here's a correct implementation of **findMatchingObject:**:

```
- findMatchingObject:anObject
{
    id match;

    match = nil;
    while (match == nil) {
        NSAutoreleasePool *subpool = [[NSAutoreleasePool alloc] init];

        /* Do a search that creates a lot of temporary objects. */
        match = [self expensiveSearchForObject:anObject];

        if (match != nil) [match retain]; /* Keep match around. */

        [subpool release];
    }
```

```
    [match setIsMatch:YES forObject:anObject];

    return [match autorelease];   /* Let match go and return it. */
}
```

By retaining **match** while **subpool** is in effect and autoreleasing it after the **subpool** has been released, **match** is effectively moved from **subpool** to the pool that was previously in effect. This gives it a longer lifetime and allows it to be sent messages outside the loop and to be returned to the invoker of **findMatchingObject:**.

Instance Variables

None declared in this class.

Method Types

Adding an object to the current pool	+ addObject:
Adding an object to a pool	– addObject:
Debugging an autorelease pool	+ enableDoubleReleaseCheck: + enableRelease: + setPoolCountThreshold:

Class Methods

addObject

+ (void)**addObject:***anObject*

Adds *anObject* to the active autorelease pool in the current thread, so that it will be sent a **release** message when the pool itself is deallocated. The same object may be added several times to the active pool, and will receive a **release** message for each time it was added. *anObject* must not be **nil**. This method is equivalent to:

```
[anObject autorelease];
```

enableDoubleReleaseCheck:

+ (void)**enableDoubleReleaseCheck:**(BOOL)*enable*

When enabled, **release** and **autorelease** invocation checks to see if this object has been released too many times. This check is performed by searching all pools, and makes programs run very slowly. By default, **enableDoubleReleaseCheck:** is set to NO.

enableRelease:

+ (void)**enableRelease:**(BOOL)*enable*

By setting **enableRelease:** to NO, **release** and **autorelease** messages are effectively ignored, allowing all objects to remain in memory. Note that this will cause your use of memory to increas. By default, **enableRelease:** is set to YES.

setPoolCountThreshold:

+ (void)**setPoolCountThreshhold:**(unsigned)*trash*

This method aids in debugging autorelease pools; when the pool size reaches a multiple of *trash*, this method will call a well-known method (indicated in the console). You can then set a breakpoint on that method in the debugger.

To disable **setPoolCountThreshold:** (it is disabled by default), set *trash* to 0.

Instance Methods

addObject:

– (void)**addObject:***anObject*

Adds *anObject* to the receiver, so that it will be sent a **release** message when the pool itself is deallocated. The same object may be added several times to the same pool, and will receive a **release** message for each time it was added. *anObject* must not be **nil**.