# Using Autorelease Pool Blocks

Autorelease pool blocks provide a mechanism whereby you can relinquish ownership of an object, but avoid the possibility of it being deallocated immediately (such as when you return an object from a method). Typically, you don't need to create your own autorelease pool blocks, but there are some situations in which either you must or it is beneficial to do so.

## About Autorelease Pool Blocks

An autorelease pool block is marked using `@autoreleasepool`, as illustrated in the following example:

```
@autoreleasepool {
    // Code that creates autoreleased objects.
}
```

At the end of the autorelease pool block, objects that received an `autorelease` message within the block are sent a `release` message—an object receives a `release` message for each time it was sent an `autorelease` message within the block.

Like any other code block, autorelease pool blocks can be nested:

```
@autoreleasepool {
    // . . .
    @autoreleasepool {
        // . . .
    }
    . . .
}
```

(You wouldn't normally see code exactly as above; typically code within an autorelease pool block in one source file would invoke code in another source file that is contained within another autorelease pool block.) For a given `autorelease` message, the corresponding `release` message is sent at the end of the autorelease pool block in which the `autorelease` message was sent.

Cocoa always expects code to be executed within an autorelease pool block, otherwise autoreleased objects do not get released and your application leaks memory. (If you send an `autorelease` message outside of an autorelease pool block, Cocoa logs a suitable error message.) The AppKit and UIKit frameworks process each event-loop iteration (such as a mouse down event or a tap) within an autorelease pool block. Therefore you typically do not have to create an autorelease pool block yourself, or even see the code that is used to create one. There are, however, three occasions when you might use your own autorelease pool blocks:

- If you are writing a program that is not based on a UI framework, such as a command-line tool.
- If you write a loop that creates many temporary objects.

  You may use an autorelease pool block inside the loop to dispose of those objects before the next iteration. Using an autorelease pool block in the loop helps to reduce the maximum memory footprint of the application.
- If you spawn a secondary thread.

  You must create your own autorelease pool block as soon as the thread begins executing; otherwise, your application will leak objects. (See Autorelease Pool Blocks and Threads for details.)

## Use Local Autorelease Pool Blocks to Reduce Peak Memory Footprint

Many programs create temporary objects that are autoreleased. These objects add to the program's memory footprint until the end of the block. In many situations, allowing temporary objects to accumulate until the end of the current event-loop iteration does not result in excessive overhead; in some situations, however, you may create a large number of temporary objects that add substantially to memory footprint and that you want to dispose of more quickly. In these latter cases, you can create your own autorelease pool block. At the end of the block, the temporary objects are released, which typically results in their deallocation thereby reducing the program's memory footprint.

The following example shows how you might use a local autorelease pool block in a `for` loop.

```
NSArray *urls = <# An array of file URLs #>;
for (NSURL *url in urls) {

    @autoreleasepool {
        NSError *error;
        NSString *fileContents = [NSString stringWithContentsOfURL:url
                                         encoding:NSUTF8StringEncoding
error:&error];
        /* Process the string, creating and autoreleasing more objects. */
    }
}
```

The `for` loop processes one file at a time. Any object (such as `fileContents`) sent an `autorelease` message inside the autorelease pool block is released at the end of the block.

After an autorelease pool block, you should regard any object that was autoreleased within the block as "disposed of." Do not send a message to that object or return it to the invoker of your method. If you must use a temporary object beyond an autorelease pool block, you can do so by sending a `retain` message to the object within the block and then send it `autorelease` after the block, as illustrated in this example:

```
– (id)findMatchingObject:(id)anObject {

    id match;
    while (match == nil) {
        @autoreleasepool {

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

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

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

Sending `retain` to `match` within the autorelease pool block the and sending `autorelease` to it after the autorelease pool block extends the lifetime of `match` and allows it to receive messages outside the loop and be returned to the invoker of `findMatchingObject:`.

## Autorelease Pool Blocks and Threads

Each thread in a Cocoa application maintains its own stack of autorelease pool blocks. If you are writing a Foundation-only program or if you detach a thread, you need to create your own autorelease pool block.

If your application or thread is long-lived and potentially generates a lot of autoreleased objects, you should use autorelease pool blocks (like AppKit and UIKit do on the main thread); otherwise, autoreleased objects accumulate and your memory footprint grows. If your detached thread does not make Cocoa calls, you do not need to use an autorelease pool block.

**Note:** If you create secondary threads using the POSIX thread APIs instead of `NSThread`, you cannot use Cocoa unless Cocoa is in multithreading mode. Cocoa enters multithreading mode only after detaching its first `NSThread` object. To use Cocoa on secondary POSIX threads, your application must first detach at least one `NSThread` object, which can immediately exit. You can test whether Cocoa is in multithreading mode with the `NSThread` class method `isMultiThreaded`.