Dynamic Method Resolution

This chapter describes how you can provide an implementation of a method dynamically.

Dynamic Method Resolution

There are situations where you might want to provide an implementation of a method dynamically. For example, the Objective-C declared properties feature (see Declared Properties in *The Objective-C Programming Language*) includes the `@dynamic` directive:

```
@dynamic propertyName;
```

which tells the compiler that the methods associated with the property will be provided dynamically.

You can implement the methods `resolveInstanceMethod:` and `resolveClassMethod:` to dynamically provide an implementation for a given selector for an instance and class method respectively.

An Objective-C method is simply a C function that take at least two arguments—`self` and `_cmd`. You can add a function to a class as a method using the function `class_addMethod`. Therefore, given the following function:

```
void dynamicMethodIMP(id self, SEL _cmd) {
    // implementation ....
}
```

you can dynamically add it to a class as a method (called `resolveThisMethodDynamically`) using `resolveInstanceMethod:` like this:

```
@implementation MyClass
+ (BOOL)resolveInstanceMethod:(SEL)aSEL
{
    if (aSEL == @selector(resolveThisMethodDynamically)) {
          class_addMethod([self class], aSEL, (IMP) dynamicMethodIMP, "v@:");
          return YES;
    }
    return [super resolveInstanceMethod:aSEL];
}
@end
```

Forwarding methods (as described in Message Forwarding) and dynamic method resolution are, largely, orthogonal. A class has the opportunity to dynamically resolve a method before the forwarding mechanism kicks in. If `respondsToSelector:` or `instancesRespondToSelector:` is invoked, the dynamic method resolver is given the opportunity to provide an `IMP` for the selector first. If you implement `resolveInstanceMethod:` but want particular selectors to actually be forwarded via the forwarding mechanism, you return `NO` for those selectors.

Dynamic Loading

An Objective-C program can load and link new classes and categories while it's running. The new code is incorporated into the program and treated identically to classes and categories loaded at the start.

Dynamic loading can be used to do a lot of different things. For example, the various modules in the System Preferences application are dynamically loaded.

In the Cocoa environment, dynamic loading is commonly used to allow applications to be customized. Others can write modules that your program loads at runtime—much as Interface Builder loads custom palettes and the OS X System Preferences application loads custom preference modules. The loadable modules extend what your application can do. They contribute to it in ways that you permit but could not have anticipated or defined yourself. You provide the framework, but others provide the code.

Although there is a runtime function that performs dynamic loading of Objective-C modules in Mach-O files (`objc_loadModules`, defined in `objc/objc-load.h`), Cocoa's `NSBundle` class provides a significantly more convenient interface for dynamic loading—one that's object-oriented and integrated with related services. See the `NSBundle` class specification in the Foundation framework reference for information on the `NSBundle` class and its use. See *OS X ABI Mach-O File Format Reference* for information on Mach-O files.
