12 Creating Frameworks and Dynamic Shared Libraries

Setting up a framework project

Making a header file private

Installing a precompiled header

Providing backward compatibility

Adding public API

Verifying compatibility between two libraries

My library was dukedom large enough.

Shakespeare, *The Tempest*

I shall sleep, and move with the moving ships
Change as the winds change, veer with the tide.

Algernon Charles Swinburne

Since 'tis Nature's law to change, Constancy alone is strange.

Hon Wilmot, Earl of Rochester, *A Dialogue between Strephon and Daphne*

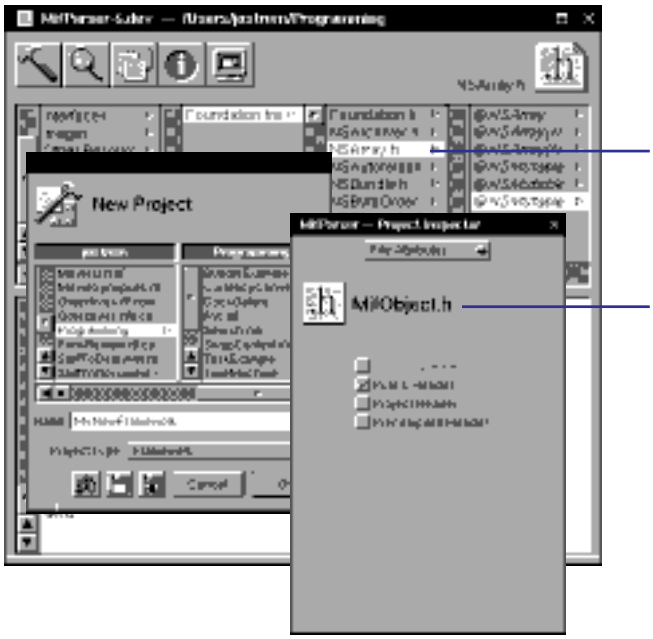
Setting up a framework project

1 **Create a project with Framework as the project type.**

2 **Add header (.h) and implementation (.m) files to the project.**

3 **Specify which header files, if any, should be private.**

A framework is a bundle containing a dynamic shared library. Both Framework and Library project types build dynamic shared libraries. The difference is that frameworks bundle the library file with its headers, documentation, and resources.



A project linked against a framework has easy access to headers.

You create a framework or library project from the New panel, just like any other project, but you must do some additional set-up using the Inspector and the makefiles.

The Framework vs. the Library

Because of their convenience, you'll want to create framework projects instead of library projects in most cases. However, if the project doesn't use resources and doesn't contain API that is public to your users (for example, if you distribute an application that uses a private library), you may choose to create a library project instead. If you need to create a static library (and you shouldn't need to), you must create a library project instead of a framework.

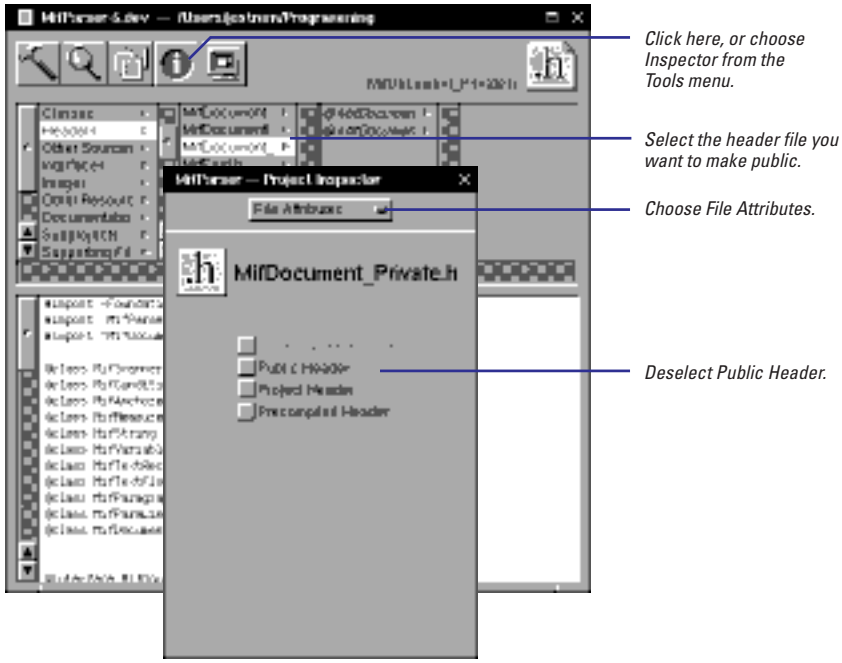
Creating a library project is very similar to creating a framework project. The tasks described in this chapter are things you do when you create either type of project. The main differences between creating a library project and creating a framework project are:

- The name of the binary file. For library projects. The name is **lib***ProjectName*.*MajorVersion*.**dylib**. For framework projects it is just *ProjectName*.
- Publishing header files. For framework projects, all header files are public by default. In Library projects, header files are private by default. To install them so that the library's users may access them, you must use the File Attributes inspector to mark each header file as public, and you must specify where to install them using the **PUBLIC_HEADER_DIR** macro in the file **Makefile.preamble**.

Making a header file private

1. **Select the header file in the project browser.**
2. **Click the Inspector button.**
3. **Choose File Attributes in the Inspector panel.**
4. **Deselect Public Header in the Inspector panel.**

By default, all of a framework's header files are public. When the framework is installed, the headers are installed in the framework's **Headers** subdirectory and the framework's users can see those headers from their projects. If you have a header that you don't want your users to see, you must mark it as private.



A good reason to make a header file private is to make sure your users don't use the API. This frees you to change it later. See "Tips and Tricks to Changing the Major Version" in this chapter.

Setting the Search Path for Frameworks and Libraries

When you link a program with a framework (or library), the framework binary's full path is recorded in the program executable. By default, a program only looks in that one location for the binary. If it can't find it, the program won't launch.

To have a program look in more than one location, set the environment variable **DYLD_LIBRARY_PATH**. This variable works like the **PATH** environment variable. For example, if you enter the following commands in a Terminal window, the Foo application will look for the binary file **MyFramework** in two locations: the recorded location and in the directory **~/Library/MyFramework.framework**.

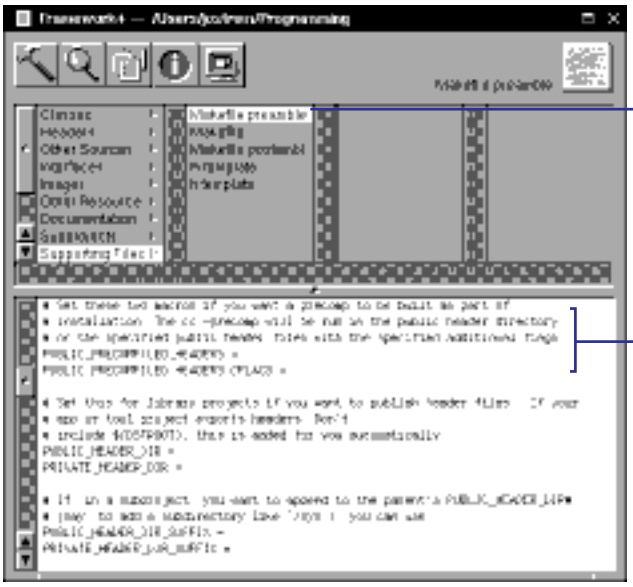
```
% setenv DYLD_LIBRARY_PATH \
~/Library/MyFramework.framework
% Foo.app/Foo
```

Installing a precompiled header

1 **Select the Makefile.preamble file under Supporting Files.**

2 **Set the macros that affect the precompilation of a header after installation.**

You might want to install a precompiled header file so that your users' projects compile faster. Installing a precompiled header is different from creating a precompiled header for a project because a header must be precompiled in its final location. When you create a precompiled header for a project, the header is compiled before the rest of the project. To install a precompiled header, you must first build and install the project in its destination, then precompile the header.



To precompile a header after it is installed, set these macros in **Makefile.preamble**:

Preamble Macro	Description
PUBLIC_PRECOMPILED_HEADERS	The names of the headers (**.h** extension) that should be precompiled after they are installed.
PUBLIC_PRECOMPILED_CFLAGS	The flags besides **-precomp** to pass to **cc** when precompiling.

Chapter 8 describes how to precompile a header for internal use by a project and the things to consider when you create the header file that is going to be precompiled.

Macros for the Makefile Hacker

The files **Makefile.preamble** and **Makefile.postamble** define several macros that affect frameworks and libraries. Using these macros, you can change the way a framework or library is built or installed. (See Chapter 9 for a description of the other macros in these files.)

By default, a framework project builds a bundle named *ProjectName*.**framework** with the subdirectories **Headers**, **Resources**, and **Versions**. Each major version is installed in a subdirectory under **Versions** along with its public headers, documentation, and resources in the appropriate subdirectories. Also under **Versions** is a subdirectory named **Current**, which contains links to the latest version. The subdirectories immediately under *ProjectName*.**framework** are really just symbolic links into **Current**.

A library project creates a binary file named **lib***ProjectName*.*MajorVersion*.**dylib** and a symbolic link to this file named **lib***ProjectName*.**dylib**. Both are installed in **/usr/lib**. No headers are installed by default.

Makefile.preamble Macros

SECTORDER_FLAGS Arguments to the linker's **-sectorder** option. See the **ld(1)** man page for more information.

OTHER_PUBLIC_HEADERS Header files that should be installed as public other than those marked as public in the File Attributes inspector.

OTHER_PRIVATE_HEADERS Header files that should be installed as private other than those included in the project.

PUBLIC_HEADER_DIR Location in which to install public headers. You must define this for library projects if you want header files to be installed when the library is installed. For frameworks, any header file marked as public is placed in the **Headers** subdirectory.

PUBLIC_PRECOMPILED_HEADERS Header files to be precompiled after installation. See "Installing a precompiled header" in this chapter.

PUBLIC_PRECOMPILED_HEADERS_CFLAGS See "Installing a precompiled header" in this chapter.

PRIVATE_HEADER_DIR Location in which to install private headers, which can be stripped away separately from your product build image. The default is not to install private headers.

PUBLIC_HEADER_DIR_SUFFIX Define this macro if a framework or library has a subproject whose public headers should be installed in a subdirectory of the parent's public header directory. For example, if you define this macro as **/sys**, they are installed in **Headers/sys**.

PRIVATE_HEADER_DIR_SUFFIX The same as **PUBLIC_HEADER_DIR_SUFFIX**, but for private headers.

LIBRARY_STYLE If STATIC, builds a static archive library (**.a** extension) rather than a dynamic shared library.

BUILD_OFILES_LIST_ONLY If YES, links the object files in the project together but does not call **libtool** to create a dynamic shared library from the object files. This macro is useful if you want to use the modules in another, larger library project.

Makefile.postamble Macros

CURRENTLY_ACTIVE_VERSION If YES, a symbolic link to the framework's binary file is created in the directory **Versions/Current**. If NO, the link is not created. The default is YES. Set this to NO if you want to install a new version of a framework but you still want projects to link against the previously installed version. This macro does not affect library projects. Using this macro is the same as checking the current version box on the Project Attributes inspector.

DEPLOY_WITH_VERSION_NAME This is the same as changing the version name in the Project Attributes inspector. See "Providing backward compatibility" in this chapter.

CURRENT_PROJECT_VERSION The minor version number. See "CURRENT_PROJECT_VERSION: For That Extra Level of Checking" in this chapter.

COMPATIBILITY_PROJECT_VERSION The compatibility version number. See "Adding public API" in this chapter.

DYLIB_INSTALL_NAME The name of the binary file that is built. The default is **lib***ProjectName*.*MajorVersion*.**dylib** for library projects, *ProjectName* for frameworks.

DYLIB_INSTALL_DIR Sets the path recorded in the library's binary file. **$DYLIB_INSTALL_DIR/$DYLIB_INSTALL_NAME** is passed as the argument to the **-install_name** option of **libtool**, which is used to set the name recorded in the library file to be something other than its path name. The default is not to use this option.

LIBRARY_STRIP_OPTS Options to pass to **strip** for statically linked libraries. You shouldn't have to create a static library, so you shouldn't have to use this macro.

DYNAMIC_STRIP_OPTS Options to pass to **strip** for framework projects and dynamic shared library projects.

Providing backward compatibility

1 **Click the Inspector button.**

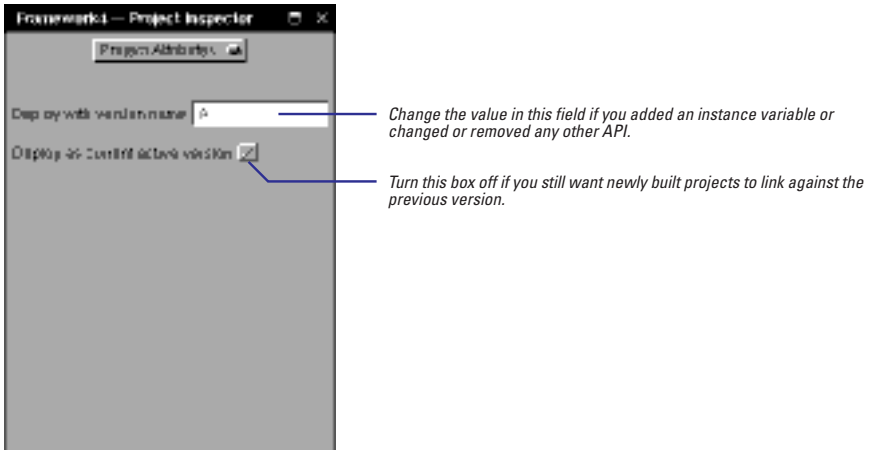
2 **Choose Project Attributes from the Inspector panel.**

3 **Set the Deploy with version name field in the Project Attributes inspector if you have removed or changed API.**

4 **Build the project.**

When you change a framework, you want to make sure not to break existing programs. If you do one of the following to your framework, you are in danger of breaking programs that link with it:

- Remove any public API.
- Change any API, such as a method or function declaration or a class name.
- Add instance variables to a class.
- Rearrange the order of instance variables in a class.
- Remove any of the architectures the framework is built for.



Whenever you make one of these changes, you should increment the framework's major version letter and provide both the new and old binary to your users. That way, programs linked against the older version of the framework will still run. New programs or modified programs will link with the newer version of the framework.

A dynamic shared library's name contains the major version letter. This name is recorded in the executable when a program links with the library. Thus, any program that links with a dynamic shared library knows that library's major version. The program won't launch if it can't find a library with the correct name.

For example, if the program **MyProg** links with version A of the framework **Misc**, the path **/LocalLibrary/Frameworks/Misc.framework/Versions/A/Misc** is recorded in **MyProg**. Suppose you add an instance variable to a class in **Misc** and change version to B. This builds **Misc.framework/Versions/B/Misc** but leaves **Versions/A/Misc** intact. Because version A still exists, **MyProg** can still run. If you change **MyProg** and rebuild it, it links with version B.

The Install in: field of the Build Attributes inspector provides the first half of the framework's full name. Variables such as **$(HOME)** are expanded *before* the path name is recorded. For more information on the Build Attributes inspector, see Chapter 9.

Tips and Tricks to Changing the Major Version

If you don't change the framework's major version number when you need to, programs linked with it will fail in unpredictable ways. If you change the major version number and you don't need to, you're cluttering up the system with compatible frameworks. You can avoid errors in changing the major version number if you follow a few simple tricks.

Don't Do It

The first trick is to avoid having to change the version number in the first place. Some ways to do this are:

- Pad classes and structs with reserved fields. Whenever you add an instance variable to a public class, you must change the major version number because subclasses depend on a superclass's size. However, you can pad a class by defining an unused instance variable of type **id**. Then, if you need to add instance variables to the class, you can instead define a whole new class containing the storage you need and have your reserved instance variable point to it.
- Don't publish API unless you want your users to use it. You can freely change private API because you can be sure no programs are using it. Declare any API in danger of changing in a private header. See "Making a header file private" in this chapter.
- Don't delete things. If a method or function no longer has any useful work to perform, leave it in the API for compatibility purposes. Make sure it returns some reasonable value. (Even if you add additional arguments to a method, leave the old form around if at all possible.)
- Remember that if you *add* API rather than change or delete it, you don't have to change the major version number because the old API still exists. The exception to this rule is instance variables. (You do have to change the compatibility version number, however. See "Adding public API" in this chapter.)

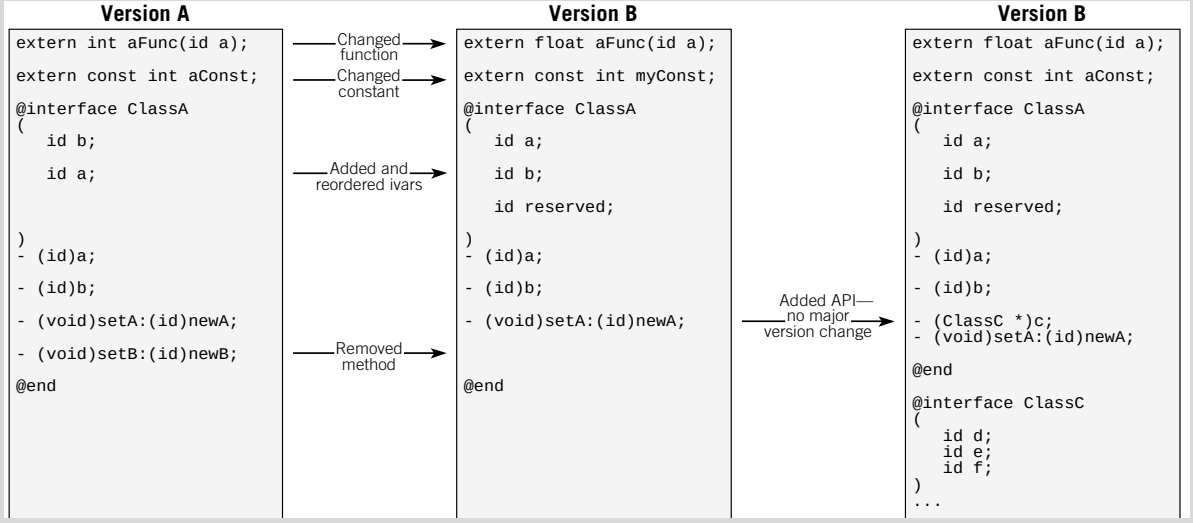
If You Do, Don't Clean It

make clean deletes the entire **.framework** bundle in the project directory, which means it deletes the old binaries in addition to the current binary. The subsequent build creates only the current version. You have no way of retrieving the earlier versions.

If you must perform a **make clean**, you'll need to create multiple copies of the project: one that builds the current version, and one for each of the previous versions. The projects that build the previous versions should set the **CURRENTLY_ACTIVE_VERSION** macro to NO so that the pointer to the current version is not changed when these older versions are installed. When you install, you'll need to install all versions.

Verify Whatever You Do

Use **cmpdylib** to make sure you did the right thing. If **cmpdylib** says the older library defines symbols that aren't defined in the newer library, you need to change the major version number. See "Verifying compatibility between two libraries" in this chapter.

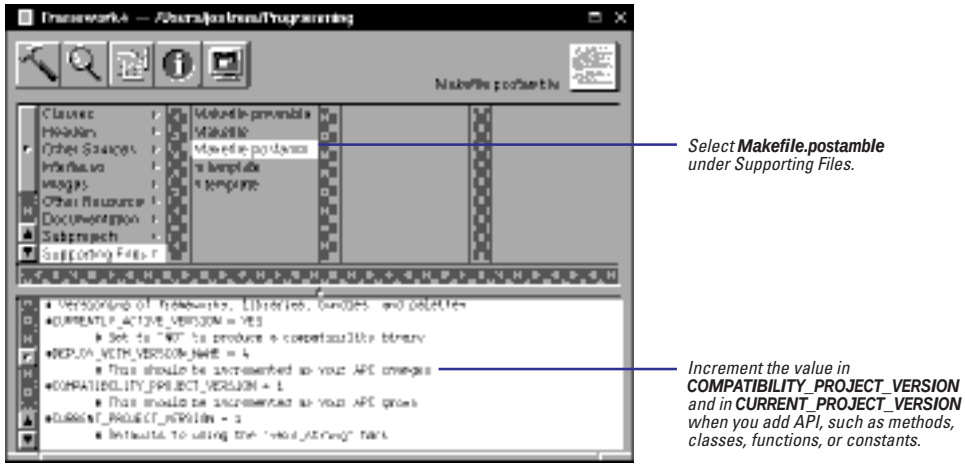


Adding public API

1 **Change the compatibility project version number and the current project version number in Makefile.postamble.**

2 **Build the project.**

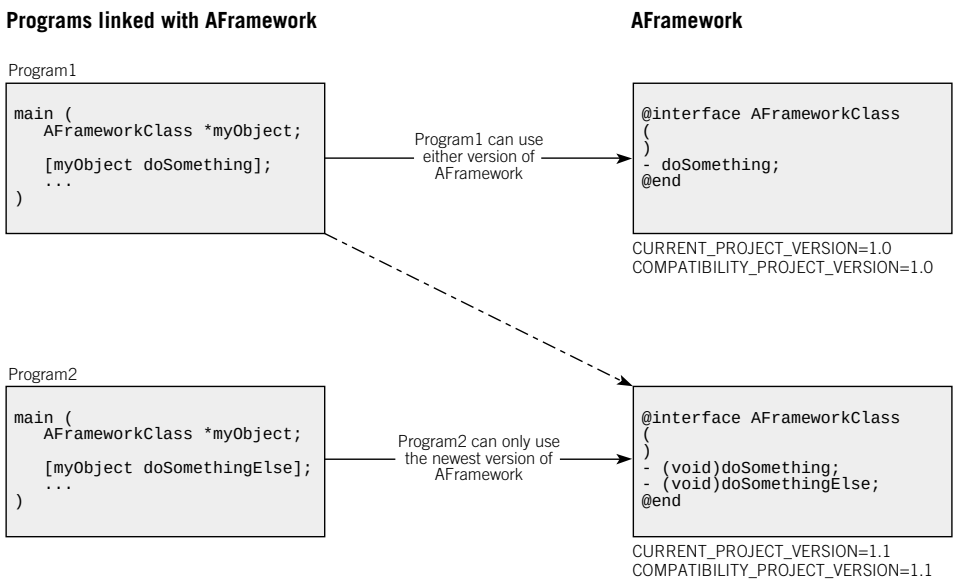
You shouldn't change the major version number when you add API (for example, if you add a class, add a method to an existing class, or add a function or constant). Adding API doesn't break existing programs. Existing programs are guaranteed to be using the older API and will still run because you've left the older API intact. However, new programs might use the new API, and therefore shouldn't try to run against older versions of the framework, which don't define that API.



When you add API, increment the compatibility version number. The compatibility version number protects programs linked with newer versions of a library from running with older versions of the library. In order for a program to launch, the compatibility version number of the framework it runs with must be equal to or greater than the **CURRENT_PROJECT_VERSION** number of the framework it linked with.

Remember that adding instance variables to a class is an incompatible change, which means you should change the version name instead of the compatibility version number. See "Tips and Tricks to Changing the Major Version" for an explanation.

Increment the value in **CURRENT_PROJECT_VERSION** whenever you change the compatibility version number. See "CURRENT_PROJECT_VERSION: For That Extra Level of Checking" in this chapter.



Why shouldn't you just change the major version number when you add API? Because programs linked with the previous version of the framework still run with the new version. If you change the major version number, the previous version remains installed on your users' systems. By changing the compatibility version number instead, you can install just one version.

CURRENT_PROJECT_VERSION: For That Extra Level of Checking

In addition to the major version number, and the compatibility version number, a dynamic shared library has a third version number. This is the minor version number or current version number. You set the current version number in the macro **CURRENT_PROJECT_VERSION**, which is in **Makefile.postamble**.

At the very least, increment **CURRENT_PROJECT_VERSION** every time you increment **COMPATIBILITY_PROJECT_VERSION**. The **CURRENT_PROJECT_VERSION** stored in a program's executable is compared with the **COMPATIBILITY_PROJECT_VERSION** stored in the library's binary file. The version in the program must be greater than or equal to the version in the library for the program to launch.

The intent is that you increment **CURRENT_PROJECT_VERSION** every time you distribute the framework when you haven't changed or added API. For example, if you fix a bug in the way a method works, you increment **CURRENT_PROJECT_VERSION**. Changes involving implementation only are almost always compatible. Programs linked against older versions of the framework can run against the new version and in fact are actually intended to run against the new version. Programs linked against the new version can still run against the old version (even though they will then encounter the bug that you have fixed).

In rare cases, someone may write a program that needs a fix from a certain version of the library. That program can use the function **NSVersionOfRunTimeLibrary()** to determine the current version of the library and take the appropriate action if the version isn't the one it needs: put up an alert panel, disable some feature of the program, or disable the entire program. Because of these rare cases where a program may need to check the version number, you should always increment **CURRENT_PROJECT_VERSION** when you distribute a new framework.

Verifying compatibility between two libraries

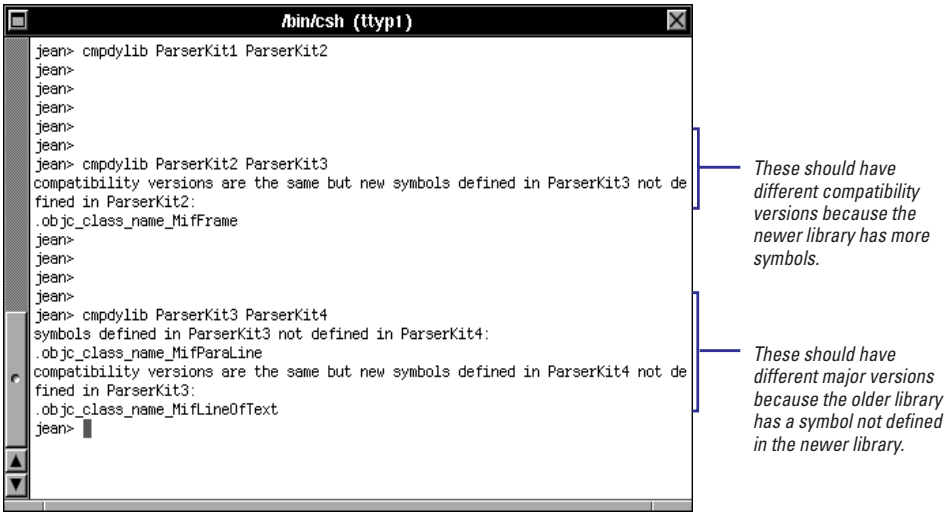
1 **Start up the Terminal application.**

2 **Perform the cmpdylib command.**

cmpdylib is a verification tool that you can use to make sure you've made the right choices about version numbers. The syntax is:

cmpdylib *oldLibName newLibName*

If *oldLibName* and *newLibName* are compatible, this command returns nothing. If they aren't compatible, it tells you why.



cmpdylib considers two libraries compatible if:

- They are built for the same architectures.
- *oldLibName* defines a subset of the symbols that *newLibName* does.
- *newLibName* defines symbols not in *oldLibName* and has a different compatibility version number.

The two libraries are incompatible if:

- They are built for different architectures.
- *oldLibName* defines symbols that aren't in *newLibName*.
- *newLibName* defines symbols not in *oldLibName* and has the same compatibility version number.

Currently, **cmpdylib** only checks C-level API and does not distinguish between public and private API. For example if you add a method, **cmpdylib** won't detect the change. Also if you change a private class, **cmpdylib** will report the change as an incompatibility.