
Macintosh
Java™
Report

目次

第1章　Macintosh における Java プログラミング

- Java はマルチプラットフォームに対応したソフトウエアを作成可能なソフトウエアの実行環境です。ネットワークに強いライブラリを持つなど、1つのまとまったソフトウエア実行環境が OS を超えて構築されています。
- この章では、Java についての一般的な知識に加えて、Java が Macintosh 環境でどのように利用できるのかを説明します。
- Java のメリット、これまでの経緯、そして現在何に注目されているかもまとめておきます。Java の開発で知っておきたい基本的なことも説明しましょう。
- Macintosh 環境での Java の実行方法や開発環境について、MRJ や SDK の内容、そして開発ツールの比較などから解説します。
- Macintosh で開発する意味についても最後にまとめておきます。

1-1　Java の特徴と利点

Java についての一般的なことをひととおりまとめておきましょう。Java はどのようなメリットがあり、なぜこれほど注目されているかを解説します。これから Java をやってみたいという方や、現状を知りたい場合には目を通して下さい。すでに基本的なことをご存知の場合は、1-4 節「Mac OS の Java 実行環境」からご覧になられると良いでしょう。

Java とは何か

Java という単語が示している本来の意味はプログラミング言語です。つまり Java は C や C++の流れをくむプログラミング言語なのですが、昨今のニュースなどで扱われる Java という言葉は、もっと広い意味で使われています。Java というプログラミング言語をベースにして作成されるソフトウエアの利用形態全般を指して、現在では「Java」と呼んでいると言って良いでしょう。Java は単にアプリケーションを作る 1 つの言語という枠組みを超え、異なる OS の上でも同一のソフトウエアが機能するなど、従来にはない機能をもったソフトウエア環境なのです。その意味では、OS であるとも言えます。既存の OS の上に構築された独自の OS でもありますし、JavaOS として独立した OS も開発されています。

最初はインターネットで注目された

Java が本格的に世の中に登場したのは、1996 年の初頭です。Java 開発の素材や実行環境のおおもととなる JDK 1.0 のリリースが、96 年の 1 月なのです。その時期、Java は Web ブラウザで動くソフトウエアであるアプレットにフォーカスしていました。Web というシステムは、世界中の文書をリンクしてつなぐなど、従来のペーパーによる文書とは異なる独自の特徴を持ち、こうした文書をブラウザさえあれば、OS やパソコンの機種に関係なく参照できました。また、文書自体は HTML として、テキストで記述できる手軽さもあります。しかし、Web の本来の機能だけを使った文書は、文字やグラフィックスをレイアウトする機能しか利用できません。当然、HTML 文書としての作成機能にはあきたらず、たとえば動きを付けたり、ユーザーの操作を受け付けるなど、もっといろいろなことを Web という基盤をもとにやりたくなってきま

す。そのためにいろいろな仕掛けが考えられたのですが、その中でも Java は汎用的なプログラミングが可能なことで、大きな可能性に期待が集まりました。一言で言えば、Web ブラウザで表示した文書内で、ソフトウエアが機能するわけです。

♦インターネットでの“動くページ”からは脱皮している

ただし、97 年頃からは、整備されていないライブラリから来る Java のシステムとしての完成度の低さがネックとなり、また Shockwave、Flash といった、インタラクティブで動きのある Web ページをより作りやすくするシステムが次々と登場したことによって、このような用途での注目は現在では浴びていません。Flash や Shockwave は Web ブラウザではプラグインが必要ですが、OS あるいはブラウザにバンドルされる傾向にあり、Web 上でのマルチメディアプレゼンテーションの標準プラットフォームになりつつあります。

このような状況を背景に Java は当初注目されるきっかけとなったインターネットの分野にとどまらず、もっと広い範囲で利用できるのではないかと期待されてきました。たとえば、業務システム開発や、あるいは組み込みシステムなどでの利用に注目が集まっています。また、分散オブジェクト開発のツールとしても Java は有力視されています。

つまり Java はプログラミング言語なのですが、見方を変えれば Java は OS そのものです。既存の OS に組み込まれた独自の OS なのです。既存の特定の OS を超えて Java という独自の OS で機能するソフトウエアを作るということが可能なのです。もはや Java はインターネットという枠組みを超えて大きく広り新たな発展をし始めていると言えるでしょう。

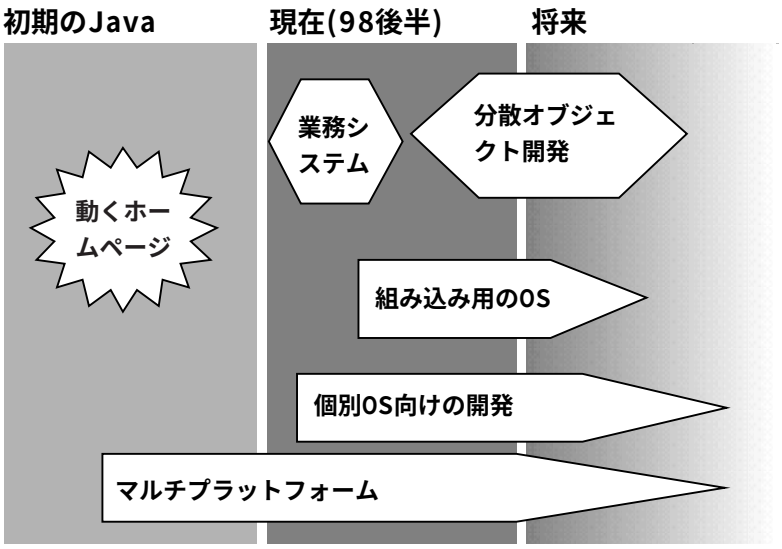

図 1-1　　Java **の役割の拡大**

言語としてみた Java の利点

まず、Java をプログラミング言語として見てみましょう。Java をプログラミング開発言語として見た場合、従来の開発言語とどこが違い、どんな点にメリットがあるかを検討しましょう。

Java は C++をベースに開発されたもので、基本的な記述方法は、C あるいは C++と共通な部分があります。特に変数や繰り返し、条件分岐などの基本的な部分はほとんど同じと言ってよいでしょう。そのため、C や C++の経験者にとって、Java はそれほど違和感がありません。

Java プログラミングの特徴を一言で言えば、オブジェクト指向プログラミングを全面的に取り入れているということでしょう。

Cはオブジェクト指向プログラミングの機能はほとんどないと言えるのですが､C++はオブジェクト指向プログラミングの中心的な開発言語となっており、開発ではもっともよく利用されています。しかしながら、C++は拡張を繰り返したために、高機能ではあるものの、非常に複雑で難しい言語になってしまいました。Java は C++よりもずっとシンプルでわかりやすい構成になるように改良されています。その反面 C++よろは機能がないと言うこともできますが、C++にあって Java にはない機能を補いう補う包括的な機能もあり、Java は C++と概ね同等のことができると言えます。また、シ

ンプルな言語体系の方が理解しやすく、また、プログラム自体も筋の通ったものになりやすい傾向があります。こうした言語としての分かりやすさ、そこから派生して生産性の高い言語であるというのが Java の特徴です。

♦何でもオブジェクトとして扱う

Java の特徴は、データを一般的に「オブジェクト」として扱うことです。オブジェクトは言い換えれば構造を自由に定義できる柔軟なデータです。逆に言えば、Java ではオブジェクトとして定義しないとプログラム中でのデータの利用は非常にやりにくくなります。オブジェクトとして一元的にいろいろな形式のデータを保管したり処理をするというのがともかく基本になります。

Java では C 言語での「ポインタ」というデータはないのですが、「オブジェクトへの参照」というデータを変数に代入しておくことはできます。

C では、「ポインタ」というデータのアドレスを利用して、データにアクセスするような手法がよく利用されていました。しかし、ポインタはプログラムを分かりにくくすると同時に、誤って関係のない部分をアクセスして書き直し、暴走する原因にもなっていました。しかしながら、Java はそうした心配はありません。

Java のオブジェクト参照は、物理的にはオブジェクトを保存したメモリへのアドレスということになるのですが、Java ではオブジェクトへの参照は演算対象になりません。オブジェクトへの参照は他の変数に代入することしかできないので、ポインタ演算のようなことはできません。オブジェクトへの参照が保存されているということを概念的に理解していればいいのです。

Java にも配列はありますが、C や C++と使い方は異なり、オブジェクト的に扱えるようになっています。

こうした特徴をメリットと見るか、デメリットと見るかはもちろん意見が分かれるでしょうが、やはりメリットと見るべきでしょう。分かりやすく安全なソフトウエアにつながるからです。しかしながら、オブジェクト指向を全面的に採用しているために、うまくオブジェクトを定義しないと、プログラム自体が混乱するということもあります。

♦オブジェクトとして定義されたライブラリ

Java でソフトウエアを組む場合、言語本来の機能だけを使って組むわけではありません。Java に限らず OS をベースにしたプログラム開発では、OS の機能やライブラリなどを利用したプログラミングが必要です。したがって、プログラミングにあたっては Java の文法うんぬんよりも、ライブラリの使い方の知識の方が膨大になり、必

要となります。

Java でもやはりライブラリが用意されています。特にコアになる基本的なライブラリは、Java の供給元である Sun Microsystems より供給されており、標準化されています。これら Java で使えるライブラリは、すべてがクラスとして定義されたオブジェクト指向のライブラリになっています。つまり、ライブラリを利用する上では、オブジェクト指向プログラミングをしなければならないのです。ここでも、全面的にオブジェクト指向が取り入れられていることになります。この基本的なライブラリについては、後で検討しましょう。

♦メンテナンスが不要なメモリ利用

プログラムでは何らかの形で必ずメモリを必要とします。特に C や C++ではメモリを確保し、そして解放するということを行わないと正しくメモリが利用できません。メモリの取得はともかく、確実に解放するというのは、プログラムする上ではかなり難しい作業です。処理途中のエラーなどで、確保されたメモリが使われずに放置されることは実際によくあることです。

Java ではメモリの解放という作業は不要で、全面的に自動的に行われます。取得は明示的に行いますが、不要になったかどうかはそのオブジェクトを利用しているかどうかがチェックされ、いらなくなったら自動的に解放するということを行っています。そのため、プログラマはともかく必要なときにメモリを確保しさえすれば良いのです。結果的に、プログラムは非常に作りやすくなります。

C あるいは C++でのプログラミングと、Visual Basic など BASIC 言語あるいはマクロ系のプログラミングとの 1 つの決定的な違いに、メモリ利用をプログラミングしないといけないかどうかがあります。BASIC 言語はそれが不要なために、気軽にプログラミングができるという面もあります。Java ではそうした BASIC 言語的な気軽さと、C や C++に匹敵するオブジェクト指向あるいは高性能なソフトウエアをつくり出すという両方のいいところを併せた特徴を持っていると言えるでしょう。

なお、不要になったメモリ領域を解放することは「ガベージコレクション」と呼ばれます。この作業自体は自動的に行われますが、そのために時間を使ってしまい、ソフトウエア全体のパフォーマンスが低下するというデメリットもあります。

♦スレッドによるマルチタスク

Java の基本的なライブラリには、スレッドを利用したマルチタスクの機能が組み込まれており、気軽にマルチタスク処理を組み込むことができます。これを利用して非同期処理を組み込むことができますし、アニメーションのような画像処理も比較的手

軽にできるようになっています。

実行環境としてみた Java の利点

Java をソフトウエアの実行環境としてみてみます。やはりいちばんの特徴は「マルチプラットフォーム」でしょう。Java のデビューのときにもっとも注目されたのがこの点で、同じソフトウエアが Windows でも Mac OS でも、UNIX でも実行できるということです。

♦クロスプラットフォームでのバイナリ互換

C 言語で標準ライブラリだけを使ったプログラムなら、若干修正の必要はありますが、どの OS でも機能します。しかしながら、Windows でコンパイルして作った EXE ファイルは、Mac OS や UNIX では実行できません。また逆も同様です。

Java のソフトウエアは、Windows で作った実行可能なファイルを、Windows ではもちろん、Mac OS でも UNIX でもそのまま実行することができます。「プログラムのソースに手を加えずに、各 OS ごとにコンパイルして実行プログラムを作成できる」というレベルではないのです。Mac OS で作った実行プログラムでも、原理的には Windows でも UNIX でも機能するというのが Java で言う「クロスプラットフォーム」という特徴なのです。このような特徴を「バイナリ互換」とも呼んでいます。

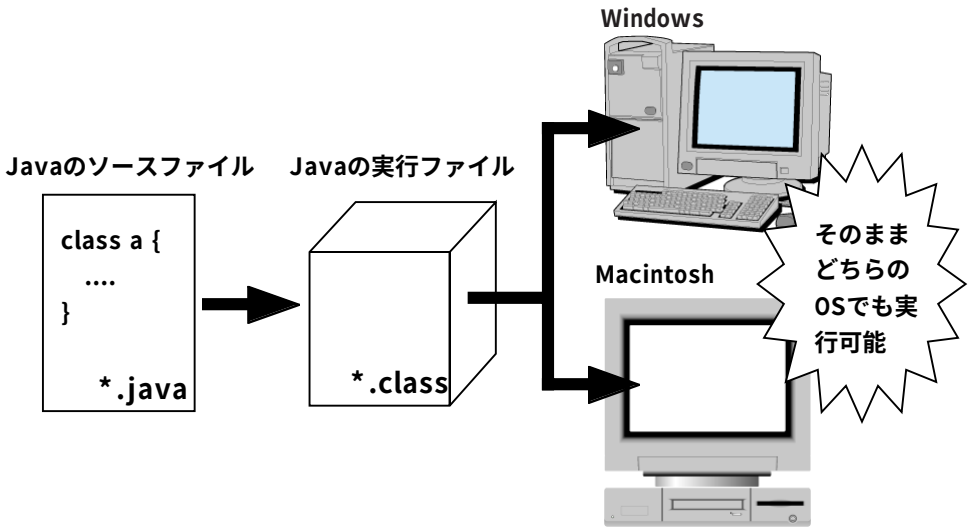

図 1-2　Java **におけるバイナリ互換**

通常、CPU や OS が違うと、実行可能なバイナリの形式も異なり、共通には使えません。では Java ならなぜできるかと言えば、簡単に言えば、OS や CPU とは別に共通の「Java 実行環境」を構築しているからです。つまり、Windows や Mac OS には、Java を実行するためのコンピュータが内蔵されています。そのコンピュータはソフトウエアで実現されており、バーチャルマシン（VM：Virtual Machine）あるいは VM などと呼ばれています。

この VM が OS に内蔵されていれば、同一のバイナリファイルが、どの OS ででも実行できるのです。もちろん、各 OS に VM が用意されていないと Java は実行できませんが、現実には Windows も Mac OS も、VM をブラウザメーカーあるいは OS メーカーによって積極的に構築されており、結果的に世の中に存在するほとんどのパソコンで、Java の VM が機能しているという状況になっています。

また、VM そのものに加えて、各 VM の実行環境で、クラスライブラリが提供されています。VM とセットにして提供されるべき基本的なクラスライブラリがしっかり定義されており、それが十分に OS 機能を果たしているため、Java は OS に依存しない共通の基盤を持った OS として機能することになるのです。

なお、VM は完全に 1 から構築されたものではなく、部分的に各 OS の機能を利用します。たとえば、Java のソフトウエアがウィンドウを表示したりキー入力を受け付けるときには、実際には背後で稼動している OS の機能を流用します。つまりクラスライブラリの見えない部分で OS の機能を利用しているのです。このように Java には稼動している OS の機能を利用するメカニズムも用意されています。ただし、一般のプログラマはその部分を扱う必要はなく、クラスライブラリの機能だけを見ていればいいでしょう。

ちなみに先程触れたように、Java はデビュー時には Web ブラウザでの利用にフォーカスされていたため、Web ブラウザ自体に VM が組み込まれました。Netscape Communicator/Navigator では、ブラウザ自身が用意した VM しか利用できません。Internet Explorer はブラウザ向けに用意したものと OS に組み込んだものを切り替えることができます。Netscape の VM のうち、Macintosh 版は Ver.4.5 でも最新版 Java のライブラリ機能にすべては対応していません。Netscape もいずれは OS の VM を利用するような形態になる予定です。

♦VM の問題点

Java は VM 上で機能することから、OS に依存しないソフトウエアを、しかもバイナリ互換で作成できるのが大きなメリットです。しかし、逆にこのことがデメリット

にもなっています。VM を実現するには、Java プログラムのバイナリコードを実行する CPU をエミュレーションすることになり、どうしても実行速度は遅くなります。つまり、バイナリーコードを実際に CPU が実行するのではなく、搭載している CPU 上のプログラムを使ってあたかも Java 対応 CPU が機能しているかのように見せ掛けているわけです。

ただ、VM であるために、通常のアプリケーションなどに比べて実行速度が低かった点については年々解決されてきています。1 つの技術は「JIT（Just In-Time）コンパイラ」と呼ばれるもので、Symantec 社の製品が有名です。これは、Java のバイナリーコードを実行直前に、稼動している CPU 向けに書き直して、そのパソコンで動いている CPU のコードに翻訳して実行するというものです。実際には CPU 本来のバイナリで実行するので、スピードは速くなります。たとえば、Mac OS だと、Java のバイナリコードが、PowerPC の実行可能なコードに翻訳されて実行するわけです。最近の VM には JIT も標準で組み込まれるようになり、実行速度は目に見えて改善されています。ただし、JIT の場合は、実行前にコードを翻訳する時間が必要になり、CPU があまり高いパフォーマンスでない場合には、ソフトウエアの起動に時間がかかるようになってしまいます。

さらに、99 年には、HotSpot と呼ばれる記述が実用化される予定です。これは言うならば動的コンパイラであり、実行時間がかかる処理を順次稼動している CPU のコードにコンパイルするような処理を実現します。また、メモリー利用や複数のタスクの実行機能などに改良が加えられ、とにかく効率の良いソフトウエアの処理が実現するはずです。

近い将来、このようにエミュレーションで動く VM であるために実行速度が遅いという Java の現状のデメリットは、解決されると期待して良さそうです。

♦動的なリンクにより軽量ソフトウエアを実現

Java で作成したソフトウエアは、動的なリンク機能により、実行するときに必要なライブラリを探して結合することができます。C や C++でも動的ライブラリは不可能ではないのですが、一般にはそうした機能は OS の機能などとして提供されています。通常、コンパイラとしてはあらかじめライブラリを取り込んで、実行ファイルを作ります。

Java は、コアになるクラスライブラリについては、コンパイルしてバイナリを作成するときには一体化しません。実際に実行する段階で、その場にあるライブラリの本体を呼び出して結合し、実行します。従来のアプリケーションでは、ライブラリを実

行ファイルに含める必要があったため、シンプルなアプリケーションでもそれなりに大きなサイズになっていました。しかしながら、Java ではライブラリをいっしょにしなくてもいいので、シンプルなものは非常に小さいサイズのファイルで事が足ります。

こうした機能はインターネットを経由して取り込まれたアプレットの実行において、ダウンロード時間の節約になるというメリットにもつながります。いずれにしても、重複したライブラリ利用は避けることができるというわけです。

安全で強固な環境

先に Java の言語としての特徴として、「ポインタ」がないことを説明しました。C 言語のようにポインタ演算でデータの位置を決める場合、バグやプログラマが想定していない状況が起きた結果、データ範囲外を読み書きしてしまう可能性が出てきます。そうすると、ソフトウエアの動作が不安定になることがあります。Java 言語ではデータの範囲外をアクセスするようなプログラミングはほとんど不可能なので、関係ないところを読み書きするようなプログラムは書くことができません。これだけでも、データやロードしたソフトウエア自体を保護する機能が働いていると言えるでしょう。

また、Java の実行環境では「セキュリティ」という考え方が導入されています。VM の動作として実行可能な範囲を制限できるのです。たとえば、ファイルの読み書きの処理は、Web ページに組み込まれるアプレットではできない処理です。仮にできるようになっていれば、Web ページを読むと、ハードディスクの中身を書き換えるというアプレットも作成できますが、悪意を持ったソフトであれば被害も大きくなります。また、悪意はなくても動作が不安定だと何をするか分かりません。そこで Web ブラウザで開いた Web ページにあるアプレットは、そのページ内、あるいは同一サーバーに対してのみ処理ができるという制約が付けられています。こうした制約を施すというセキュリティがあるために、Java で作成したアプレットは安全なソフトウエアとして機能することが保証できるのです。

1-2　Java で開発できるソフトウエア

Java ではいろいろな形態のソフトウエアが作成可能ですが、現状では、アプリケーションとアプレットというのが一般的な形です。最近はサーバーで機能するサーブレットも作成できるようになってきています。また、コンポーネントとしての JavaBeans も注目を集めています。これらについて説明をしましょう。

Web ページに埋め込むアプレット

アプレットは Java の登場時から存在するソフトウエアの形態で、Web ページに埋め込んで利用されるものです。以下に説明するように、従来にはない形態のソフトウエアであったために、Java に注目が集まったと言っても過言ではありません。

アプレットは、拡張子が.class というファイルに保存された Java の実行形式のバイナリファイルを、Web ページ内で機能させたものです。なお、アプレットは.class ファイルにあるのが一般的ですが、アーカイブファイルの.jar ファイルに存在することもあります。Web ページは HTML テキストでその内容を記述できますが、APPLET タグを利用して、アプレットを Web ページに埋め込みます。見かけ上は JPEG などのグラフィックスと同じように長方形の範囲を確保し、その中でソフトウエアの実行が行われます。

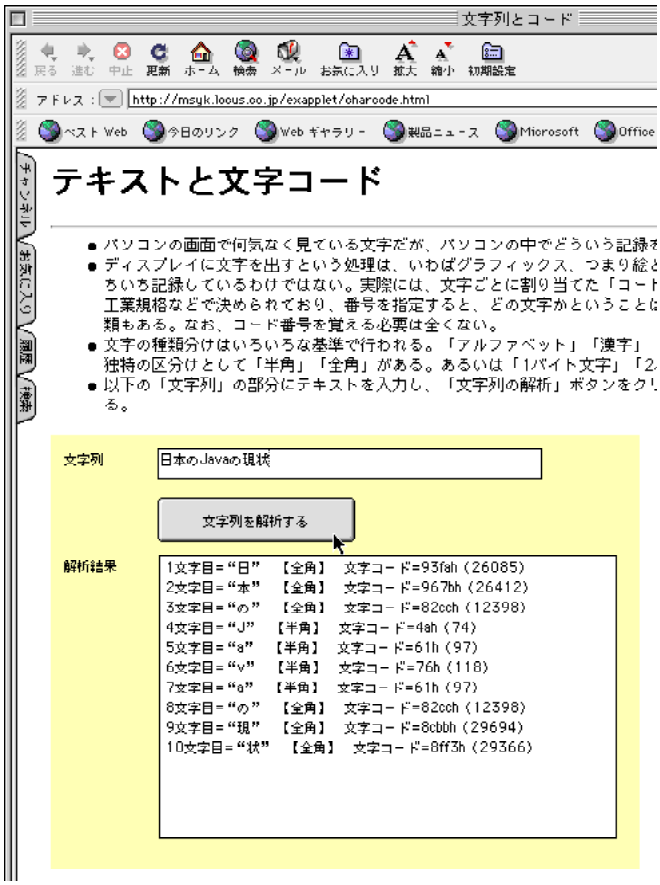

図 1-3 Web ページに埋め込まれたアプレット。下半分のボックスがアプレットで、テキスト入力枠や、ボタンなどが配置されている

アプレットの.class ファイルは、HTML ファイルと同じサーバーに置かれます。画像ファイルもサーバーに置かれますが、それと同じイメージで考えて下さい。HTML テキストの中に APPLET タグがあると、サーバーからクライアントのブラウザにアプレットがダウンロードされます。そして、Web ページを参照しているクライアントの中にある VM を利用して、アプレットが実行されます。

このとき、Web ブラウザが Mac OS で機能していても、Windows で機能していても、サーバーから同一の.class ファイルがクライアントにやってきます。VM という共通の基盤が各 OS あるいはブラウザに用意されているので、同じ.class ファイルが違う OS で機能するということになるのです。

サーバー上で公開されているアプレットは、場合によっては不特定多数のクライアントで実行されます。そこでセキュリティを確保するために、アプレットで利用できる VM の機能は制限されています。その制限はブラウザである程度はコントロールで

きますが、中にはファイル処理など絶対にできないこともあります。

また、アプレットは Java の場合、極めて簡単にプログラミングできる点も見逃せません。アプレットの原形にあたるクラスが用意されているので、そのクラスを継承し

て、望む機能を追加するだけでアプレットは作成できます。サーバーからのダウンロードするなどのややこしいことは全部 Web ブラウザと VM がやってくれます。

Java のアプリケーション

Java でも一般的なアプリケーションを作成できます。Mac OS では、ダブルクリックして実行可能なごく普通のアプリケーションを作成できます。Windows では、EXE ファイルを作成することが可能です。ただし、一般にはその OS で実行可能になっているファイルをそのまま別の OS に持って行っても実行することはできないでしょう。たとえ Java のバイナリ部分はどの OS で動いたとしても、アプリケーションとして実行させる部分はファイルに含まれることになり、それが OS ごとに違うからです。

だからといって、OS ごとにアプリケーションを 1 から作る必要があるわけではありません。Java VM は共通なので、Java のソースコードは、Mac OS、Windows 用で同一のものを利用できます。それをもとに、Mac OS 用のアプリケーションを作ったり、Windows 用のアプリケーションを作るということをすればいいわけです。ただし、1 つの開発ツールで Mac OS 用と Windows 用の実行可能ファイル作成ができるとは限らず、それぞれ別の開発環境を使わないといけないかもしれません。

なお、Java のソースコードをコンパイルした結果は.class ファイルに保存されますが、ファイルの拡張子についてはこのようにアプレットだけでなくアプリケーションも同様です。

Mac OS にも Windows でも、.class 形式のアプリケーションを実行するツールは存在します。それらのツールを利用すると、同一の.class ファイルのアプリケーションが、Mac OS でも Windows でも実行できます。

サーブレット

Web ページで、入力するテキストボックスなどがあって、閲覧者の入力を受け付けるものも見られます。そのようなページの多くは「ボタンをクリックすると入力データがサーバーに送付され、サーバー側でプログラムを実行して、入力データをデー

タベースに保存する」などの処理を行うようなメカニズムを CGI（Common Gateway Inteface）で実現しています。このような、ユーザーのリクエストによってサーバー上で実行するようなソフトウエアも、Java で作成できるようになっています。こうしたソフトウエアを「サーブレット」と呼んでいます。

サーブレットを実行するには、サーバーに VM が組み込まれていることに加えて、Web サーバーがサーブレット実行の機能を持って行なければなりません。一般には、Web サーバーがサーブレットを実行するという手順になります。もともとサーブレットを実行可能な Web サーバーもありますが、サーバーのプラグインとしてサーブレットの実行環境が提供されている場合もあります。現状ではむしろ後者の方が一般的なサーブレット実行の方法になるでしょう。

JavaBeans とコンポーネント

ソフトウエアを部品化することを「コンポーネント化」などと言いますが、Java では、JavaBeans という枠組みで、基本的なコンポーネント化の機能をサポートしています。JavaBeans に従ったソフトウエアの部品を「Bean」と呼んでいます。

通常 Bean だけでは必要な機能は満たされません。Bean を組み合わせることによって、アプレットやアプリケーションを作ります。このとき、グラフィカルな開発環境などで部品として取り扱い可能な形式は JavaBeans として仕様が定められています。開発環境で部品と部品を線でつないで、アクションに対する処理を選択するようなことを可能にするのが JavaBeans の 1 つの目的です。

さらに、たとえばサーバーで機能するソフトウエアを作るための Enterprise JavaBeans のような規格も作られています。サーバーで実行されるソフトウエアはクライアントからのリクエストに応じた形式で必要な処理を行うので、その意味ではシステム全体から見ればソフトウエアの部品のように見えるわけです。こうした部品を作る枠組みが Enterprise JavaBeans ということになります。

1-3 フレームワークとしての Java

Java を特徴付けるものとして、そのクラスライブラリがあります。クラスライブラリは誰もが自由に作ることができますが､基本になる部分は Sun Microsystems で開発され公開されています。基本部分は、いわば OS としての機能と言っても良い部分です。

作成ソフトで使う機能を提供するライブラリ

Java が初期の頃は、基本クラスライブラリの機能が、一般的な OS に比べて目に見えて貧弱でした。しかしながら、Java のリリースが進むにつれて、一般の OS にかなり近付いてきています。また、ネットワーク機能のように、一般の OS よりもより使いやすく充実した機能もあります。

こうした基本クラスライブラリは、アプリケーションやアプレットなど Java ソフトウエアを作るフレームワーク（枠組み、あるいはひな形）となります。もちろん、こうしたクラスライブラリを無視して独自に構築してもいいのですが、普通はそうはしません。与えられたクラスライブラリをいかにうまく利用するかと言うのがプログラミングの基本になります。基本クラスライブラリの内容についての概略を説明しましょう。

なお、いくつかのクラスを集めたものを「パッケージ」と呼んでいます。パッケージは概念的なものと思ってもいいのですが、実際には特定のクラスを集めるものとしてプログラム上で定義されているものです。

充実したクラスライブラリ

クラスライブラリの中には、Web ブラウザに組み込むアプレットの原形である Applet クラスが定義されています。アプレットとして機能するために必要なさまざまなものがすでに用意されています。アプレットは、このクラスを継承して作成します。極端に言えば、アプレットの中身をどうするかという記述だけを加えればアプレットは作成できてしまいます。しかも、テキストボックスやドロップダウンリストなどのユーザーインタフェースを構築するオブジェクトは、次に説明する AWT で定義されており、アプレットには簡単に追加できます。また、再描画処理も簡単に記述できます。このようにアプレット作成にはかなり最適化されているのは、当たり前と言えば

当たり前なのですが、Java の特徴としては顕著な部分です。

Java では基本的なデータとして整数の int 型などは使えるのですが、データはオブジェクトとしてある意味では一般化した使い方をします。クラスライブラリでは基本的なデータ型のオブジェクトに加えて、オブジェクトを複数まとめるような使い方をするためのクラス定義などもあり、オブジェクトを利用するための基本的な機能も提供しています。

グラフィックスとユーザーインタフェース

グラフィックス処理の中心になるのが、AWT（Abstruct Windowing Toolkit）と呼ばれるパッケージで、文字通りウィンドウ処理を行うものですが、現実にはもっとたくさんのことができると考えてよいでしょう。グラフィックス処理や基本的な GUI の一切を取り仕切るパッケージです。

まず、AWT にはウィンドウ表示のもとになる、一般的な画面表示領域を定義するクラスがあります。また、その中にオブジェクトを表示領域に簡単に加えることができるようなメカニズムを持ったクラスも定義されます。これらを利用して、テキストボックスやドロップダウンリストなどのクラスも定義されており、ユーザーインタフェースのオブジェクトを簡単に処理できるようになっています。

ただし、グラフィックス描画については、ごく基本的なことしか行なえません。複雑なグラフィック処理については、Java 2D と呼ばれる Sun Microsystems より提供されるパッケージでサポートしています。

Java のライブラリは「イベント」として、何か変化があったことを別のオブジェクトに伝達する手段を備えています。ユーザーインタフェースオブジェクトを使うときには、必ず利用することになるでしょう。たとえば、ユーザーがボタンを押したり、リストを選択したときなどにイベントが発生します。このイベントの扱いが、初期の Java と、97 年頃にリリースされた JDK 1.1 とではやや異なっています。当然、後から出た方がより高機能なのですが、Macintosh 版の Netscape では未だに新しいイベント処理が VM に組み込まれていません。ここは注意が必要なところです。

98 年頃より、Swing という名称のパッケージ群が Sun Microsystems より提供され始めており、より高度なユーザーインタフェース構築のための機能が利用できるようになってきています。AWT では限られていたオブジェクトの種類も、Swing では豊富に利用できます。正しくは、Swing は JFC（Java Foundation Classes）というライブラリの一部分で、JFC にはさらにドラッグ&ドロップの扱いなども組み込まれます。

また、マルチメディアでは、ビデオ画像などを扱う Java Media や、3D オブジェクトを扱う Java 3D なども利用されはじめています。

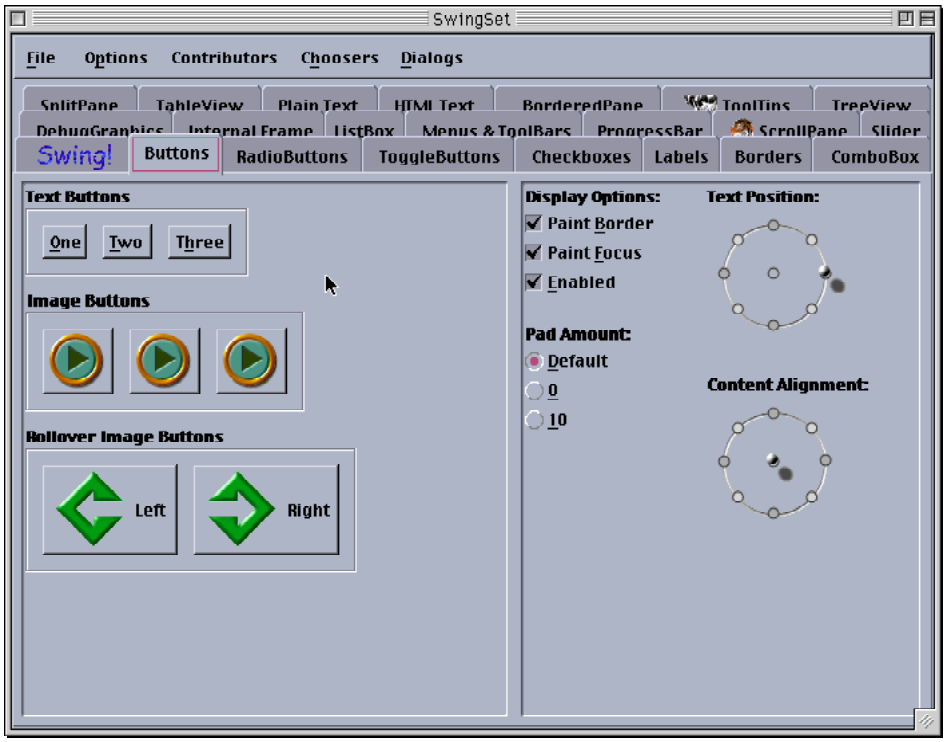

図 1-4　Swing のコンポーネントをデモンストレーションするアプリケーション

入出力処理とネットワーク

Java にはファイルの処理を行うパッケージも用意されています。このパッケージには単純なバイト単位の入出力だけでなく、行単位の入出力を行う機能なども用意されていますが、どちらかと言えば基本的な機能が中心です。ドライブの情報などもアプリケーションなどでは知りたいところですが、そうした OS に依存するような情報については標準のライブラリでは得ることができません。従ってパッケージで行なえるのは一般的なファイル処理に限られます。また、アプレットではファイル入出力がまったく利用できない点も注意が必要です。ただし、将来的には適切なセキュリティをかけた上で、アプレットでもファイル処理ができるようになるとされています。

Java のライブラリを、一般の OS から見たときに大きく特徴付けているのがネットワーク関連のパッケージでしょう。Java が注目され始めた当初はアプレットによって Web ページで動く画面が作成できる点が強調されていましたが、TCP/IP をベースに通信するソフトウエアを簡単に作成できるという隠れた面も見逃せません。そのこと

が Java がネットワークに強いと評価される大きな要因です。TCP/IP のソケットを簡単に用意でき、しかも、行単位の入出力などもできるので、電子メールを送付するようなプログラムくらいなら簡単に作成できます。

また、電子メールで言えば、JavaMail というパッケージにより、電子メールのやりとりも比較的簡単にプログラミングができます。ネットワーク、特にインターネットを使うプログラムとなると、Java はだんぜん強味を発揮するようになります

データベース環境

データベースにアクセスするためのパッケージとして、JDBC（Java DataBase Connectivity）が用意されています。JDBC 用のドライバを利用すれば、Java のソフトウエアからデータベースエンジンにアクセスできます。SQL 言語レベルの処理はもちろん、テーブルのデータをオブジェクトとして扱って処理するようなことも可能です。

分散オブジェクト

Java は分散オブジェクト環境での開発も視野に入っており、特に大規模な業務システムでの利用も始められているようです。オブジェクト間の通信では、CORBA をベースにすることもできるようになってます。また、RMI（Remote Method Invocation）として、別のマシンにあるオブジェクトのメソッドを呼び出すようなメカニズムも、分散オブジェクト環境を実現するのに使われています。そして、JavaIDL というパッケージで、CORBA などのオブジェクトリクエストブローカー（オブジェクト間通信を行う基本システム）を利用した分散オブジェクト構築までが可能になっています。

JDK について

JDK は Java Development Kit の略で、Sun Microsystems によって開発された Java 開発のための基本となる素材を集めたものです。JDK が Java 環境の正式なリリースで、その中には、開発ツール、基本クラスライブラリ、ドキュメントなどが含まれています。JDK はそのバージョンとともに記述されるのが一般的で、バージョンによりサポートされている機能が異なります。いずれにしても、JDK というのは、Java 開発の中ではよく出てくる言葉です。

JDK には基本クラスライブラリが含まれており、JDK を入手することによって、クラスライブラリが手に入ります。従って、そのクラスライブラリを使ったプログラ

ムの作成もできます。あるいは、配付されたクラスライブラリを利用するソフトウエアの実行もできるようになるというわけです。

Java 開発ツールのメーカーは、JDK をライセンスされており、自社の開発ソフトに JDK としてリリースされたクラスライブラリを含めることができます。しかしながら、一般には JDK に対応した開発ソフトが発売されるのは、その JDK がリリースされてから時間が経過した後です。Sun Microsystems 以外のメーカーより供給される開発ツールを使いながらなおかつ JDK も入手する理由として、新しくリリースされるクラスライブラリをなるべく早く手に入れたいということが挙げられます。

また、JDK には Java のコンパイラなど、開発に必要なツールが含まれています。市販の開発ツールを利用するまでもなく、JDK だけでプログラムを実行可能な形式にすることができます。Sun の Web ページなどからダウンロードできるので、事実上フリーの開発ツールとして JDK が存在すると言えば良いでしょうか。

♦コンパイラ以外のツールもある

JDK はコンパイラだけでなく、いろいろなツールを含んでいます。アーカイブファイルの処理をするものもありますし、分散オブジェクト開発用のツールなどもあります。さらに、作成したプログラムの解説ドキュメントを作成する JavaDoc と呼ばれるツールも JDK に含まれます。クラスライブラリの解説ドキュメントを見たことがあるかもしれませんが、プログラム中のクラス定義から、そのクラスにあるメソッドなどの使い方を解説した HTML 文書を作成する機能もあります。

♦Macintosh 版は、SDK としてリリース

初期の頃は、Macintosh 版の JDK もありましたが、現在は、Apple がリリースする MRJ SDK（1-5「Mac OS での Java 開発環境」を参照）がそれに取って変わっています。Sun は、Solaris 版と Windows 版の JDK をリリースしており、それとは別に、Sun と Apple が協力して、JDK 相当の MRJ SDK をリリースしています。MRJ については後で詳しく説明しましょう。

1-4　Mac OS での Java 実行環境

Mac OS では MRJ という Java の実行環境が OS に統合されて提供されています。MRJ について説明しましょう。また、Mac OS で動作する MRJ 以外の Java VM についても説明します。

MRJ

Mac OS Runtime for Java（MRJ）は、Apple が提供する Java の実行環境です。MRJ は OS に統合される形で Java の VM を提供しており、これが Mac OS での代表的な VM であると言えるでしょう。正式版としては、Ver.2.0 がリリースされていますが、Early Release として Ver.2.1 の配付が 98 年 5 月より始まっています。Early Release は一種のベータテスト的なもので、開発途中の状態で公開することで、対応ソフトの開発を促がそうという意味あいがあります。略して「EA 版」あるいは「MRJ Ver.2.1ea2」のように呼ばれます。後者は、MRJ の Ver.2.1 の Early Release 2 版であるという意味です。

Ver.2.0 の稼動環境は、Mac OS 8 以降で、8MB 以上の空きメモリが必要となっています。ただし、System 7.6.1 以上でもカスタムインストールで必要なシステム機能を組み込めば機能することになっています。Ver.2.0 は、PowerPC でも 68040 でも利用できます。

Ver.2.1 の Early Release 版は、System 7.6.1 以上と Ver.2.0 より古いシステムも対応 OS に含めていますが、PowerPC のみの対応となっています。さらに、32MB 以上の RAM 搭載機で、仮想記憶をオンにして、メモリを 33MB 以上にしておくという条件を満たさなければ鳴りません。

♦MRJ のインストール

MRJ のインストールは、一般的なインストーラを使って行なえます。Early Release 版はおそらくはインターネットを通じて入手できるでしょう。MRJ 2.0 については、Mac OS 8 あるいは 8.1 には、CD-ROM の「Mac OS 特別付録」フォルダの中にある「Macintosh Runtime for Java」フォルダの中にインストーラがありますので、それをダブルクリックしてインストールします。

Mac OS 8 以上で使うときには、そのままインストールを行えばよいでしょう。MRJ Ver.2.0 を System 7.6.1 にインストールするときには、Appearance Manager など、カス

タムインストールで必要な機能拡張ソフトも同時に組み込む必要があります。何を組み込めばいいかはインストーラと一緒に配付されるドキュメントで必ずチェックしてください。

インストールした結果、システムフォルダの「機能拡張」フォルダに、MRJLibrariesフォルダが作成されます。実行に必要なファイルがそのフォルダに納められます。

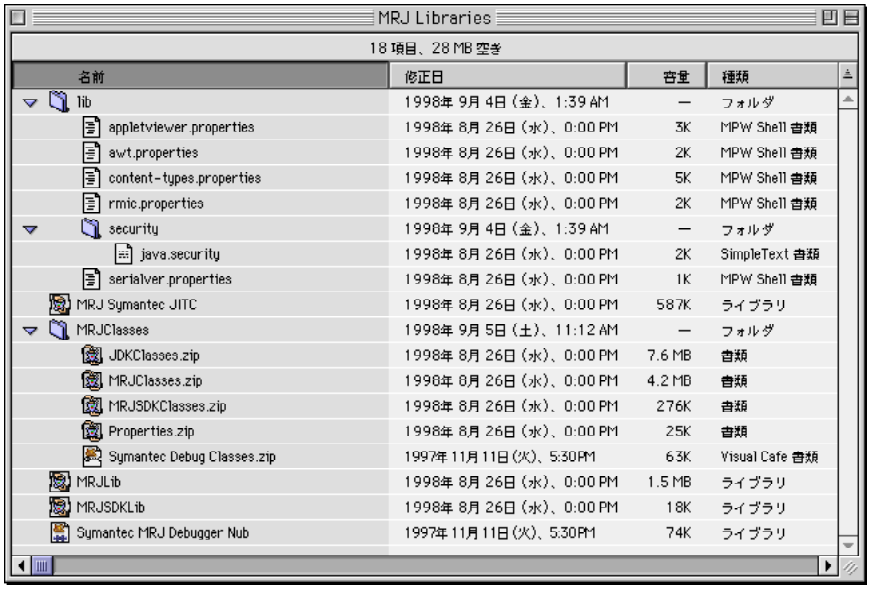

図 1-5　インストールされた MRJLibraries フォルダの内容

なお、MRJ 2.1 では MRJLibraries フォルダの中身以外にも、機能拡張フォルダのMRJSubPorts というファイルも実行に必要になります。

♦Java のソフトウエアを実行する

MRJ はいくつかの主要部分がありますが、Java の VM を実現するライブラリが最も重要になります。これはおそらく、MRJLibraries フォルダの MRJLib というファイルがその役割を持っているものだと思われます。このファイルは種類が「ライブラリ」です。たとえば、Internet Explorer が Java アプレットを実行するときに MRJ を使う場合には、内部的に MRJLib を呼び出して Java VM の実行環境を整え、そこから Java のアプレットを呼び出しているものと思われます。また、ダブルクリックして実行可能なアプリケーション形式の Java ソフトウエアでは、このライブラリを利用するようなプログラムがアプリケーションに組み込まれる模様です。つまり、アプリケーションの起動時にライブラリをロードして、Java VM の実行環境を整えた上で、Java プログラムの実行を始めるのです。

Java のアプリケーションの多くは、Java のクラスライブラリを使いますが、MRJ では、MRJLibraries フォルダにある MRJClasses フォルダの中にある.zip ファイルを参照することになります。MRJ は実行時に、クラスライブラリの存在するフォルダをこの MRJClasses フォルダであると認識しています。JDKClasses.zip には、JDK で提供される基本的なクラスがアーカイブされ圧縮されて入っています。Java の VM は圧縮してアーカイブしたようなファイルからも、その中身のクラスを利用できます。MRJClasses.zip には、MRJ だけで提供されるクラスが含まれています。

以上が、Java ソフトウエアを実行するのに必要なシステム機能です。ファイルの数は少ないのですが、いずれも MB サイズの非常に大きなファイルとなっています。

♦複数バージョンの MRJ を切り替える

いくつかのバージョンの MRJ を切り替えて使いたいのであれば、「機能拡張」フォルダの MRJLibraries フォルダを差し替えることで可能です。たとえば、MRJ 2.0 をインストールしているとしましょう。そのときの MRJLibraries フォルダをどこか別のところにコピーしておきます。そして、MRJ 2.1ea2 をインストールしたとします。このとき、MRJLibraries フォルダは MRJ 2.1ea2 が入っています。この状態で、MRJ 2.0 を使いたい場合、まず、2.1ea2 の MRJLibraries フォルダをどこか別のところに移動し、あらかじめ取っておいた 2.0 の MRJLibraries フォルダを「機能拡張」フォルダに戻します。Early Release の MRJ では動作が不安定だったり、あるいは仕様変更によってそれまでに作っていたソフトウエアが機能しなくなることもあります。その場合、こうした方法で MRJ を切り替えて、Java のソフトウエアを利用したり、あるいは開発作業を行う必要が出てくるでしょう。

Internet Explorer と Microsoft の VM

Web ブラウザの Internet Explorer では、Web ページ内の Java アプレットを実行する VM を、MRJ のものと Microsoft が提供する VM とで切り替えることができるようになっています。「編集」メニューの「初期設定」を選択して、「初期設定」ダイアログボックスを表示します。左側のリストで Java の項目を選択すると、Java についての設定ができるようになっています。ここで、VM を切り替えることができます。

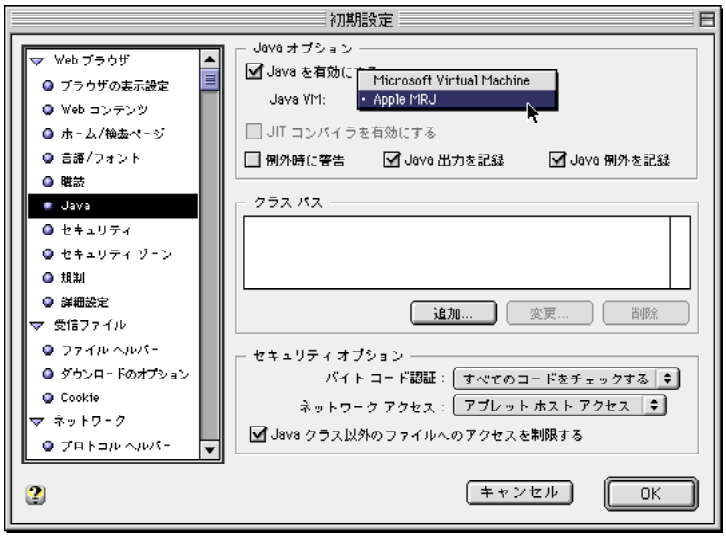

図 1-6　使用する Java VM を選択する

「Apple MRJ」を選択すると、システムに組み込まれている MRJ を利用してアプレットを実行します。

「Microsoft Virtual Machine」を選択すると、Internet Explorer 自体で用意している VM で Java のアプレットを実行します。「機能拡張」フォルダに MS Library Folder というフォルダがあり、そこに Microsoft VM Library (PPC)というファイルがあります。そのライブラリが Java VM だと思われます。さらに Classes (4.01)フォルダに、classes.ms.zip や extra.converters.zip がありますが、ここに基本のクラスライブラリが入っている模様です。

Microsoft の VM と MRJ の性能をくらべれば、MRJ の方がベンチマークテストでは高得点を出します。しかしながら、MRJ Ver.2.0 と比べると、画像やダイアログボックスのような処理は、どちらかと言えば Microsoft VM の方が高い得点になります。また、いくつかのアプレットを動かしてみた感じでは、Microsoft VM の方が安定しているように思われる場合もありました。実際には切り替えて違いを見てみる必要が出てくるでしょう。

このように、Internet Explorer を使う上では 2 つの VM がありますが、将来的には MRJ に一本化される予定になっています。Microsoft の VM の技術と MRJ を統合化するというアナウンスがすでに出ています。

♦アプレット開発時の注意

アプレットを開発しているときには、後で説明する Applet Runner を利用してデバッグなどをします。しかしながら、ある程度完成した後は、Web ページ内で、他の要

素との兼ね合いもチェックする必要が出てくるので、Web ページにおいて実行してみたいときもあるでしょう。その場合、Internet Explorer でファイルとしてアクセスできるローカルの HTML ファイルを開き、その状態でアプレットを開くことができます。HTML ページを作成するときには確認のために、HTML ファイルを Internet Explorer にドラッグ&ドロップして開いて見ますが、その方法でもアプレットは起動します。しかしながら、アプレットが存在するフォルダまでのフルパス名の中に漢字あるいは 2 バイトコードのフォルダ名が存在すると、うまくアプレットが機能せずにエラーになります。アプレットのクラスが見つからないというエラーなので、おそらくは 2 バイトコードを含んだパスの文字列を、Internet Explorer が Java VM にうまく伝達できていないのだと思われます。MRJ でも Microsoft の VM でも同じ現象になります。フォルダ名は 1 バイト文字だけにしておくなどの対処が必要でしょう。

Netscape の VM

Netscape Communicator/Navigator にも Java の VM が組み込まれていますが、これは固定されたものです。Netscape を使うときには、MRJ を VM として使うことはできません。Netscape の VM の Macintosh 版は、JDK のすべての機能をサポートしていないことに注意が必要です。JDK 1.1 で導入されたイベントモデルや JDBC などいくつかサポートしていない機能があります。そのため、JDK 1.1 が基本となっている現状では、Java アプレットの実行環境としては適していないブラウザと言わざるを得ません。

以下に説明する Java Plug-In の Macintosh 版が登場すれば、Netscape でも MRJ を使うということが可能になるかもしれませんが、Ver.4.5 の段階ではまだリリースされていません。いろいろな意味で Netscape はブラウザとしての評価は高いのですが、Macintosh 上での Java 開発あるいは実行環境として見たときには、JDK 1.1 への完全対応がなされていないことで選択肢からははずれてしまいます。

Java Plug-In による実行

Web ブラウザでの Java アプレットの実行では、Netscape のように独自に VM を包含していると、JDK の新しいバージョンへの対応が遅れるなど、実行環境の整備に手間取ることもありました。そこで、Java の実行環境自体を Web ブラウザのプラグインとして提供することも始まっています。Windows 版や Solaris 版はすでに Sun より

リリースされています。Macintosh 版は Apple からリリースされることになっていますが、98 年 9 月現在、リリースされていません。Java Plug-In は、以前は Java Activator という名前で開発されていたものです。

Java Plug-In は、Sun より配付されている JDK あるいは JRE（Java Runtime Environment：JDK のうちの Java VM の部分だけで構成されるもの）を VM として利用します。そして、Java のアプレットを使うには、プラグインを使う形式で HTML を記述するため、従来からの APPLET タグではなく、OBJECT タグで利用します。そのため、タグの書き直しを行うユーティリティも Sun Microsystems より配付されています。

Java Plug-In を使えば、Web ブラウザに搭載されている VM ではなく、別途供給する JRE を使います。JRE は JDK から Java の実行のために必要な部分だけを取り出したようなもので、通常は JDK に合わせて最新版が利用できます。VM の最新版を利用しやすいことや、ブラウザが用意する VM ごとの違いによる問題は回避できることなどから、Plug-In を前提に開発を行うということも行われ始めています。

Java コンソールについて

Java にはコンソールを実現するライブラリがあり、これを使うと行単位の入出力画面の利用が可能になっています。あるいはファイルなどを入出力とすることもできます。コンソールは、MS-DOS や昔の端末のように、行に入力したり、あるいは出力を行単位で行うと言った機能です。テキストの入出力しかできません。MS-DOS や UNIX では、標準入出力として、コンソール形式のデータ入力や出力を、プログラムの上では簡単に実現できるような仕組みが用意されています。Java のライブラリでも同様なことを実現しています。

UNIX や MS-DOS のコマンドでは、標準入出力の利用がメインになりますが、Windows や Mac OS のような GUI の OS では、マウスによるポイントやウィンドウ上のグラフィック表示などが主体になります。Java のコンソールについても、主要な入出力形態というよりも、補助的な扱いです。しかも、一般的には出力だけに利用されます。たとえば、処理途中のデータを確認するといったデバッグ的な使い方や、Java の特徴的なエラー処理である例外が発生したときのメッセージ用として利用されます。このように、コンソール出力は開発のためのデータやエラーメッセージ用に利用されるので、開発者はその機能と使い方を知っておく必要があります。

♦実際の動作でのコンソール

Mac OS では、VM ごとにコンソールの形式が違います。Web ブラウザには、Web ブラウザ自身が Java コンソールを表示する機能を持っています。通常はメニューを選択してコンソールを表示します。Internet Explorer 4.01 では、「表示」メニューの「Java メッセージ」を選択します。Netscape Navigator では、Navitagor メニュー（船の舵のアイコン）の「Java コンソール」を選択します。バージョンや対応言語によってもメニュー項目は変わります。いずれにしても、Web ブラウザではコンソールはメニュー操作をしないと表示されず、勝手に表示されるわけではありません。

一方、Web ブラウザ上ではなくアプリケーションとして機能していたり、あるいは Applet Runner でアプレットを実行していると、プログラム中でコンソールへの出力があれば、自動的にウィンドウが開かれて、そこに出力結果が表示されます。つまり、独立したコンソールウィンドウが自動的に表示されるというわけです。

1-5　Mac OS での Java 開発環境

Macintosh 環境で、Java のソフトウエアを開発する場合に、どのような素材が利用でき、利用可能な開発ツールがどのようなものかについて説明しましょう。

MRJ SDK について

MRJ SDK は Apple からリリースされている Java の開発キットで、Windows や Solaris での JDK と同等の意味を持つ、基本的な開発ツールです。Java の実行環境である MRJ と開発環境である MRJ SDK は別のものだということをまずしっかり認識しておいてください。Macintosh 向けの基本クラスのリリースも、MRJ SDK で行われますが、必ずしも実行環境の MRJ よりも先に出るとは限りません。最近は概ね MRJ の少し後に MRJ SDK がリリースされるというのが傾向です。MRJ SDK は、以下に示すように、プログラミングのための素材を提供することが主な用途だと言えるでしょう。

♦SDK の中身の概要

MRJ SDK は、インストーラを起動すればインストールができます。通常は、同一バージョンの実行環境である MRJ をインストールした上で、追加でインストールします。SDK には実行環境は含まれていません。MRJ SDK をインストールすると、「MRJ SDK 2.1」のようなフォルダが作成されます。以下、その中身を見ていくことにしましょう。

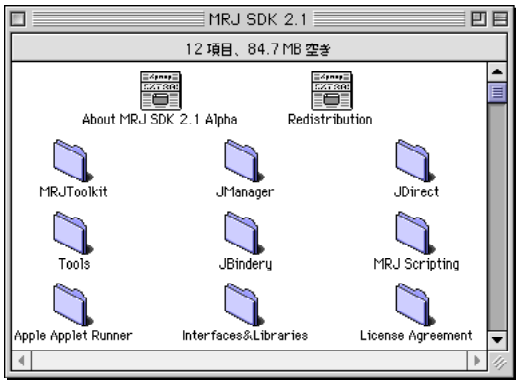

図 1-7　　MRJ SDK 2.1ea2 **の内容**

♦入手方法と現在のバージョン

本稿執筆時点（1998 年 9 月 15 日）には、正式リリース版としては、MRJ SDK 2.0.1 がリリースされています。また、早期公開されている実行環境の MRJ Ver.2.1 Early Release 2 に対して、開発環境としては MRJ SDK Ver.2.1ea2 がリリースされています。これらは、Apple の Web ページからダウンロードできます。

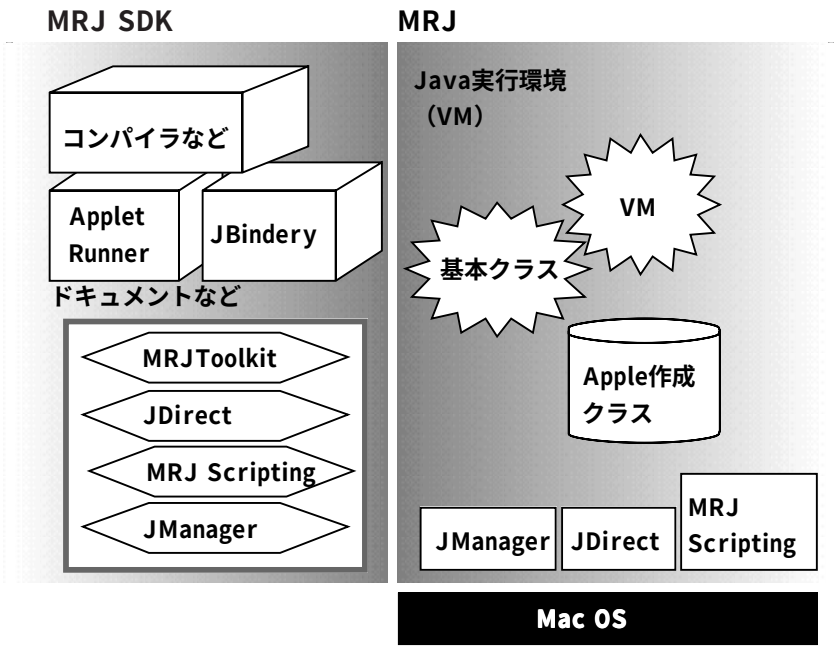

図 1-8　MRJ **と** MRJ SDK **の関係**

Apple Applet Runnner

Applet Runner は、文字通り、アプレットとして作られたソフトウエアを実行するためのアプリケーションです。JDK に含まれるものの中では、appletviewer に対応するのが Applet Runner です。これは独立したアプリケーションとして機能し、そこでアプレットを実行することができます。通常はアプレットは Web ページの中に埋め込まれて実行されますが、Applet Runner では、1 つのアプレットが 1 つのウィンドウを使って実行することができます。アプレットを実用的に使う環境というよりもむしろ、テスト的に実行したり、デバッグのためにアプレットを実行するために使います。従って、Applet Runner は開発ツールの 1 つと位置付けることができます。

Applet Runner は MRJ 2.0 までは、実行環境の MRJ といっしょに配付され、「Apple

エクストラ」フォルダ（あるいは Apple Extra フォルダ）の中の Mac OS Runtime for Java フォルダに Apple Applet Runner フォルダとしてインストールされていました。このフォルダの中に、Apple Applet Runner があります（Applet Runner として参照します）。なお、MRJ 2.1 からは、Applet Runner は、MRJ SDK のなかで配付されるようになりました。

なお、実行環境の MRJ と、Applet Runner のバージョンを合わせておかないと、うまく機能しないこともあります。たとえば、MRJ 2.0 をインストールしているときには、MRJ 2.0 に付属の Applet Runner を使います。MRJ 2.1 を使うときには MRJ SDK 2.1 にある Applet Runner を使います。開発ツールによっては、実行やデバッグ時に自動的に Applet Runner を起動しますが、このとき、いくつかのバージョンの Applet Runner が同時にハードディスクに存在すると、どれが起動するか分かりません。必要に応じて使用したいバージョンの Applet Runner 以外を削除しておく必要があるでしょう。もちろん、消してしまうのが問題な場合は圧縮をかけておき、圧縮ファイルだけを残しておくことで対処すればよいでしょう。

♦実行方法

作成したアプレットは、Java のソースコードから生成した、拡張子が.class のバイナリファイルです。.java の拡張子の付いたソースファイルから、.class ファイルがコンパイラによって作られます。このアプレットを実際に起動するためには APPLET タグを含んだ HTML 文書が必要です。アプレットを開発するときには、HTML 文書も一緒に作るのが一般的です。

Applet Runner でも、.class ファイルを直接実行するわけではありません。Applet Runner

中で、class ファイルをアプレットとして起動するタグの入った HTML ファイルを指定すると、そのアプレットを実行できるのです。Applet Runner の File メニューの Open Local HTML File、あるいは Command+O と操作して、HTML ファイルを指定してもいいのですが、Finder 上で HTML ファイルを Applet Runner にドラッグ&ドロップしてもかまいません。また、ネットワーク上のアプレットを起動するには、File メニューの Open URL（Command+N）で URL を指定して開きます。

通常、HTML ファイルの中にある APPLET タグ内で、アプレットのファイルへの相対パスがパラメータとして記述されているはずです。その記述に従って、.class ファイルのアプレットが Applet Runner によってロードされて実行されます。

HTML ファイル内に複数のアプレットの組み込みがあってもかまいません。その場

合、Applet Runner ではそれぞれのアプレットに別々のウィンドウを割り当てて実行を開始します。なお、HTML ファイルの APPLET タグ以外の部分は、Applet Runner では無視されますが、HEAD や BODY など HTML の基本的な骨格部分は、通常は HTML ファイルに含めておきます。ネットワーク上のアプレットを実行する場合も、それを組み込んである HTML ファイルを指定します。

いずれにしても、HTML ファイル自身は Applet Runner では表示されません。Applet Runner は、HTML ファイルの中の Applet タグの部分だけを取り出し、その他の部分は無視します。

♦アプレットの実行コントロール

Applet Runner の Applet メニューからは、アプレットを再ロードしたり、再起動したり、あるいは実行を停止すると言った、アプレットの実行をコントロールするメニュー項目が並んでいます。

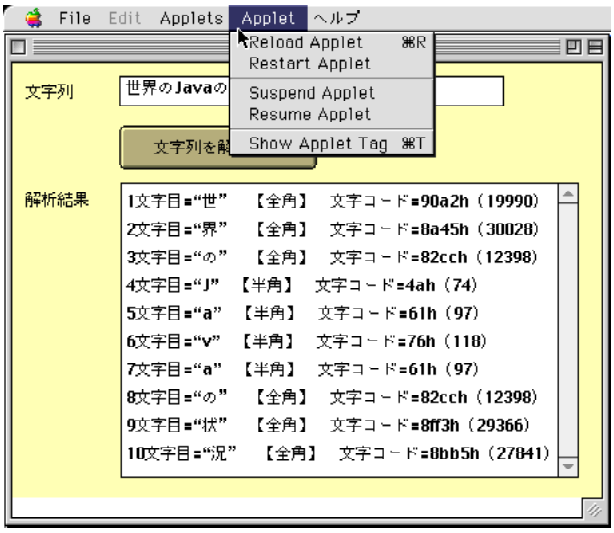

図 1-9　Applet Runner **で実行したアプレットと、**Applet **メニュー**

♦サンプルの実行方法

Applet Runner は、JDK にサンプルとして添付されているアプレットをすぐに実行できるようになっています。Applets メニューに、サンプルアプレットの名前が並んでいますが、階層メニューによって、さらにアプレットを実行する HTML 文書を選択できるようになっています。

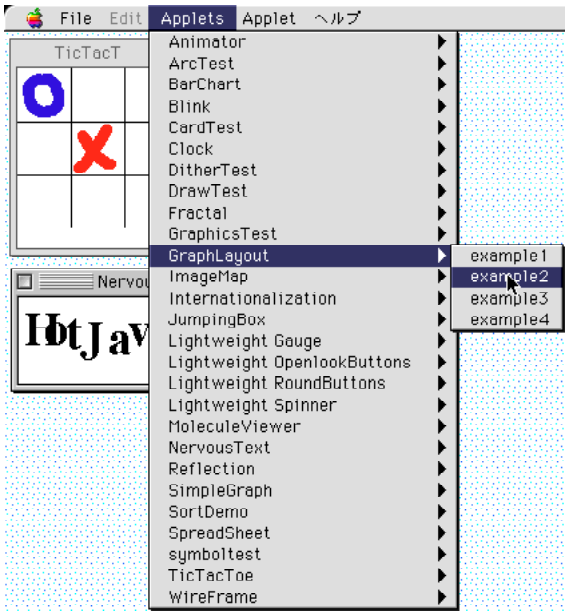

図 1-10　Applet Runner **でサンプルを実行する**（MRJ 2.0）

これらのアプレットは、Apple Applet Runner と同じフォルダの中にある Applets フォルダにあらかじめ入れられているものです。なお、MRJ SDK 2.1 の Applet Runner には、JDK のサンプルはほとんど入っていません。

♦実行環境のセキュリティに関する設定

Apple Applet Runner で、File メニューから Properties（Command+;）を選択することで、アプレットの実行環境に関する設定が行なえます。この操作で表示される以下の図のダイアログボックスは、アプレット実行環境のセキュリティについての設定が行なえるようになっています。

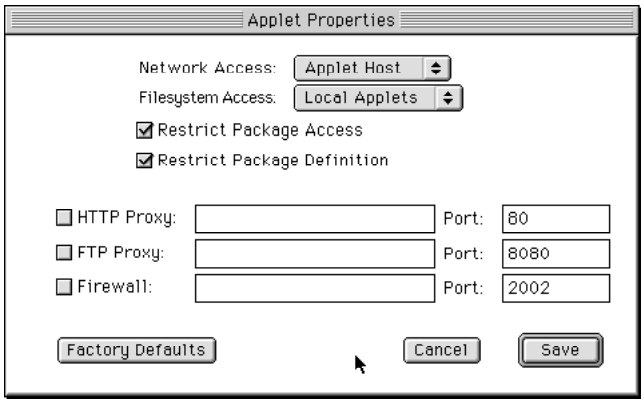

図 1-11　Applet Runner **のセキュリティ設定**

Network Access は、アプレットからアクセス可能なネットワーク上のサーバーなどについての制約です。既定値は Applet Host になっており、この状態では、アプレットが存在するのと同じホストとしか通信ができなくなっています。一般に Web ブラウザではこれと同じ制約になっているので、通常はこのままで良いでしょう。

Filesystem Access は、アプレットから利用可能なファイル処理の範囲です。既定値の Local Applets は、Applet Runner を実行しているパソコンと同じパソコンにあるアプレットを実行した場合、ファイルの処理ができることを意味しています。しかしながら、Web ブラウザでは、一般にはファイル処理は一切できないようにセキュリティが設定されています。もちろん、そうしたことを知った上で Applet Runner でファイル処理をしてもいいのですが、ともかく、ファイル処理についてのセキュリティは、Applet Runner と Web ブラウザで異なっていることは意識しておかなければならないでしょう。

チェックボックスが 2 つありますが、これらは、通常はオンにしておきます。Restric Package Access を設定しておくと信頼性の低いアプレット（開発したアプレットをネットワークを通じてダウンロードしたもの）は、Java の基本クラスライブラリにアクセスできなくなります。また、基本クラスの存在するパッケージの中に、独自に定義したクラスを含めることはできないようになっており、そうした制約を付けておくのが Restrict Package Definition の設定です。いずれも、一般的な Web ブラウザの動作に合わせて、オンにしておきます。

ほかに、プロキシサーバーやファイアーウォールの設定があります。ネットワーク環境でこれらの設定が必要な場合は、ここで設定を行います。

MRJToolkit

MRJToolkit は、実行環境である MRJ では、Java から Mac OS の機能を利用する部分を示します。MRJ SDK 2.1ea2 の段階で公開されている機能は、Finder 上でファイルをドラッグ&ドロップすることで起動可能なアプリケーションを作る、すなわち、Open Document の AppleEvent に対応可能にする機能が公開されています。他にも、ファイルタイプやクリエイターの取得および設定、リソースの処理など、Java の基本機能にはないもので、Macintosh のアプリケーションとして最低限備えておくべき機能を提供しています。実行環境としての MRJ にはこうした機能が組み込まれています。従って、MRJToolkit の機能を使うように、Java のプログラムを組めばよいということになります。

MRJ SDK には、MRJToolkit フォルダがありますが、そこには、これらの機能を Java のプログラムで使う方法を示したドキュメントと、サンプルのプログラムが入っています。サンプルのプログラムは、Metrowerks 社の開発ツールの Code Warrior で作られています。

JDirect

JDirect は、Java のソフトウエアから、C や C++で作られたライブラリなどの呼び出しを行うような機能を提供します。Mac OS のシステムコールは基本的には C の関数として定義されていますが、Java からこうした Mac OS のシステムコールを呼び出すことが JDirect では可能になります。こうしたことは、システム本来の機能を呼び出すことから「ネイティブコール」などと呼ばれています。JDirect により、Java のプログラムから、Mac OS のシステムルーチンを利用できますが、独自に C 言語などで作成したライブラリを呼び出すことにも使えます。実行環境としての MRJ に JDirect の機能が含まれています。

Java の基本ライブラリとして、JNI（Java Native Interface）という、JDirect のような機能が定義されています。Windows 版の Microsoft の VM では、良く似た名前の J/Direct という機能でネイティブコールを実現しています。このように、ネイティブコールの手法は本来の Sun の定義されたものとは違うものが出てきており、それも Sun の Microsoft への訴訟の焦点になっています。

たとえ JNI でネイティブコールが一本化されても、Mac OS 向けのソースコードは Windows では機能しませんし、逆も同様です。ネイティブな部分なので、当然共通性はないわけです。ネイティブコール方式が一本化されたとしても、ターゲットの OS ごとに API コールは違うことが一般的なので、OS ごとに別々に開発することになります。OS ごとに異なるネイティブインタフェースのシステムが機能しているからといって、根本的に開発が容易になるというわけではないのです。

実行環境としての MRJ には JDirect の機能が組み込まれています。MRJ SDK の JDirect フォルダには、ネイティブコールを行う方法を記載したドキュメントとサンプルのプログラムが含まれています。

♦Toolbox ルーチンで使うわけでもない

それでは MRJToolkit に存在しないシステム機能は、JDirect を使って利用するのでしょうか。確かにそうすることもできます。しかしながら、システムコールはハンド

ルやポインタと言った Java には存在しないデータを使います。JDirect はそれらのデータを使う工夫もありますが、うまくシステムコールを利用するように Java でプログラミングを行うというのも大変な作業になります。実際には、そうしたことをする必要がないように、MRJ は構築されている模様です。

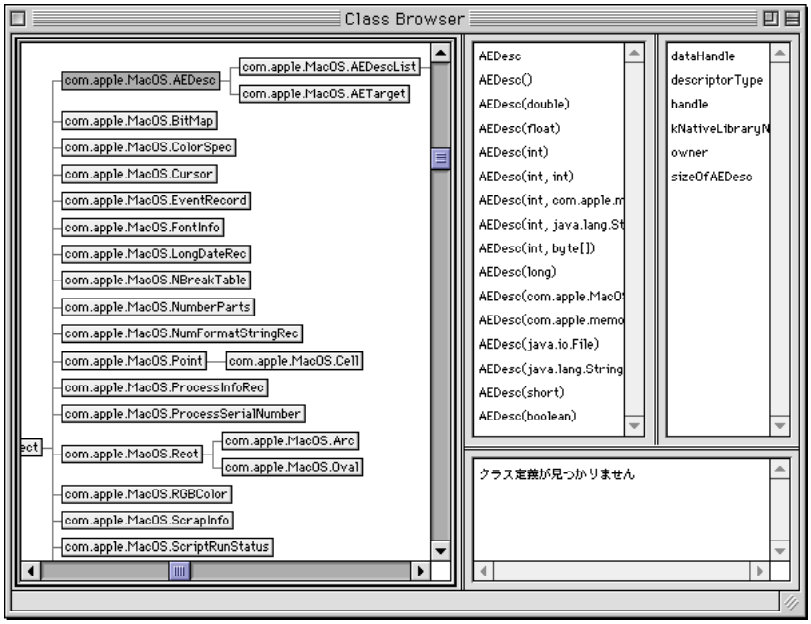

図 1-12　Visual Cafe **で見た実行環境のクラスには、システム機能を使うものが見えている**

MRJ では、Mac OS の Toolbox に用意されているシステムコールを使うためのクラスを構築してあり、それを利用することによって、Java のソフトウエアからも比較的容易に Mac OS のシステム機能を使うことができます。言い換えれば、C 言語のライブラリということを意識しなくても、Java のやり方で Toolbox の機能を利用できるのです。しかしながら、どのように使えばいいかということが一切ドキュメントになっていません。現在読んでいただいている Macintosh Java Report では、こうしたすでに構築されているクラスについての使用法を解説することを 1 つの大きな柱と位置付けています。

これら、Toolbox のシステムルーチンを直接利用するクラスでは、JDirect の手法でネイティブコールを行っている模様です。ですが、そのような部分は MRJ の実行環境として構築されており、一般のプログラマは、Apple が定義したシステムルーチンを使うためのクラスを使う方にフォーカスすることができます。つまり、Toolbox の機能を使うためのクラスを使えばそれでよいのです。その意味では、システムルーチ

ンを使うようなアプリケーションでも、JDirect を直接プログラムで使うということはほとんどないでしょう。

MRJ Scripting

MRJ SDK 2.1 から新しく加わった MRJ Scripting は、Applet Runner をスクリプト処理できるというのが 1 つの機能です。Java で作ったアプレットを、AppleScript のオブジェクトとしてコントロールすることができるのです。その場合、Applet Runner で実行しているアプレット内のボタンやテキストボックスと言った Component クラスを元にしたオブジェクト 1 つ 1 つを、AppleScript のプログラムでコントロールできます。アプレット全体のコントロールだけでなく、アプレットの中の個々のオブジェクトを AppleScript でコントロールできるという具合です。言い換えれば、Java のアプレットは、それだけですでに AppleScript 対応になるということに他なりません。

また、Java で作ったアプリケーションを AppleScript に対応させるということも、MRJ Scripting の機能です。そのための aete リソース（AppleScript の用語を管理するリソース）や対応方法を記述したドキュメントも付属します。

AppleScript も、根本は処理対象をオブジェクトとして扱い、そのオブジェクトを処理するということを目指しています。MRJScripting により、Java の世界を AppleScript から操作できるようになるわけで、AppleScript と Java の親和性も一気に高くなるでしょう。特に従来のアプリケーションでは、AppleScript つまり AppleEvent に対応するために、対応する部分のプログラムをわざわざに構築する必要があったのですが、MRJScripting がうまく機能すれば、Java のアプレットに関しては、特に AppleScript 対応のための追加プログラムをしなくても、AppleScript でのコントロールができることになります。その意味でも、AppleScript との相性が一気に良くなっていると言えるでしょう。

JBindery

JBindery は、MRJ SDK に含まれる Mac OS で機能するアプリケーションです。Java で作られたアプリケーションを Mac OS 上で実行するのが 1 つの大きな機能です。その意味では、一般の SDK の java ツールに対応したものです。また、Java で作られたアプリケーションを、Mac OS でダブルクリックして実行可能なアプリケーションに

変換するということも行います。つまり、Java のアプリケーションを Mac OS 専用のアプリケーションにしてしまうというわけです。MRJToolkit の機能などを使ってアプリケーションが適切に作られていれば、JBindery によって、たとえばテキストファイルを Finder 上でドラッグ&ドロップすると、そのテキストファイルを開いて処理をするようなアプリケーションになります。

MRJ SDK には、JBindery そのもの以外にも、その利用方法を記述したドキュメントと、サンプルの Java アプリケーションが含まれています。

♦Java アプリケーションの実行方法

Java のアプリケーションであるためには、public static void main(String args[])というスタティックなメソッドがクラス定義に含まれている必要があります。Java のソースは.java ファイルに保存されますが、コンパイル結果は.class ファイルに保存されます。そして、JBindery でその.class ファイルを実行すると、クラス内の main メソッドを探してそれを最初に実行します。main メソッド内からはプログラマが自分で作ったプログラムを実行できるように、プログラムを作成することになります。

アプリケーションとして作られてコンパイルされた.class ファイルを実行するには、class ファイルを JBindery にドラッグ&ドロップします。すると、次のようなダイアログボックスが表示されます。ここでさまざまな設定をして、Run ボタンで実行が行われます。

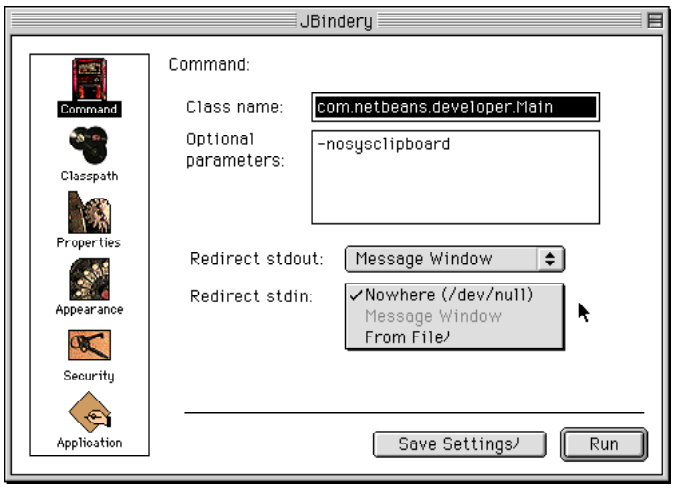

図 1-13　JBindery でアプリケーション実行の設定を行う

ダイアログボックスの左側のアイコンにより、さまざまな設定が行なえるようになっています。

Commands では、コマンドラインを指定します。実行するクラス名に加えてオプションを指定できるようになっています。また、Redirect stdout や Redirect stdin では、標準入出力を選択できます。いずれもなしにしたり、あるいはファイルを標準入出力としたり、出力ではウィンドウに出力結果を出すことから選択できます。

左のアイコンで Classpath を選択すると、クラスライブラリのファイルやファイルが存在するフォルダを指定することができます。通常は MRJLibraries/lib は使えるようになっているのでそれ以外のパスを追加するときに利用します。

Properties は、アプリケーションで利用できるプロパティを追加したりあるいは値を設定することができます。Appearance では、サイズボックスのためにウィンドウの下に領域を設けるかあるいは表示サイズのままで右下にサイズボックスを設けるかを選択します。Security は、プロキシサーバーなどの設定です。

Application は、アプリケーションを作ったときのクリエイタや、メモリのサイズ、さらにはリソースを結合する設定などが行なえます。

♦JBindery でアプリケーションファイルを作る

こうした JBindery での設定は、「Save Settings」ボタンで保存できます。ファイルの保存のダイアログボックスが表示されるので、フォルダとファイル名を指定しますが、「Save as Application」のチェックボックスをオンにしておくと、保存したものはアプリケーションとなり、Finder 上でダブルクリックして起動できます。チェックを入れないでおくと、JBindery 文書として保存され、それを Finder 上でダブルクリックすると JBindery が指定した設定で開かれて、Run ボタンにより実行ができます。つまり、実行のための設定を保存しておくということができるわけです。

JManager

MRJ の実行環境に組み込まれている JManager は、C 言語などで作られたソフトウエアから Java のソフトウエアを実行する機能を提供するものです。たとえば、Web ブラウザを開発している人や Applet Runner のようなものを自分で作るというプログラマには必要になる機能でしょう。また、ライブラリやプラグインは Java で作る、というような場合にも必要になるかもしれません。JManager を活用するとする状況としては、ソフトウエア部品としてすでに存在する Java のソフトウエアを、C や C++で作るアプリケーションで必要になるという見方もあるでしょう。

いずれにしても、通常の Java のアプレットやアプリケーションを作成しているプ

ログラマが使うような機能ではありません。Mac OS の基本機能としては重要なJManager ですが、プログラマが一般的に使う機能ではないと言えるでしょう。

MRJ SDK には、JManager の利用方法を記述したドキュメントと、C++で作られたプログラムから Java のソフトウエアを呼び出すサンプルプログラムがあります。サンプルプログラムの Java のソフトウエアを呼び出す部分を流用してプログラムに機能を組み込むのが一般的な利用方法になるでしょう。

♦Interfaces&Libraries

MRJ SDK には、Interfaces&Libraries というフォルダがあります。このフォルダには、C や Pascal のヘッダーファイルやライブラリがありますが、いずれも、ほとんどの部分は JManager の機能を使うために、開発ツールあるいはそのプロジェクトで利用するファイルです。従って、一般的な Java プログラマにとって必要性はほとんどないと思われます。

Tools フォルダの内容

MRJ SDK の Tools フォルダには、開発に必要なさまざまなツールが含まれています。特に、JDK Tools は、Windows などの他のプラットフォームでリリースされている JDK に対応するツールが含まれており、コンパイルなど基本的な開発作業ができるようになっています。

表 1-1　JDK Tools **の内容**

ファイル名	機能
javac	Java コンパイラ。Java のソースファイル（.java）をドラッグ&ドロップすれば、コンパイルして実行ファイル（.class）を生成する
jar	Java アーカイバ。Java のアーカイブファイル（.jar）を作成したり、アーカイブからファイルを取り出す
javadoc	Java ドキュメントジェネレータ。Java のソースファイルから、クラスの利用方法を記述した HTML ドキュメントを生成する。生成されたドキュメントで利用する画像ファイルは、Tools フォルダの中の javadoc images フォルダにある
javah	ネイティブコールの定義から、C のヘッダーやスタブを作成するツール
javakey	セキュリティの設定ツール。プライベートあるいはパブリックキーを設定したり、認証を行う
rmic	RMI で使用するスタブやスケルトンの作成を行う

JDK に比べていくつか存在しないツールがあります。まず、Java 実行環境の「java」や「appletviewer」はありませんが、MRJ では JBindery や Apple Applet Runner がそれに代わります。デバッガの jdb がありませんが、それに直接代わるものはありません。デバッグ作業は MacsBug でできるようにコマンド拡張ファイルは付属していますが、MacsBug はローレベルのデバッガです。MRJ SDK にはデバッグ環境はないと言ってもよいでしょう。。

MRJ SDK の Tools フォルダの中の Other Tools にある MRJ dcmd for PPC フォルダが、MacsBug 用の dcmd ファイルになっており、これによって MacsBug のコマンドを増やします。ただし、ソースコードのデバッグではありません。通常、デバッグを行うとなれば、ソースコードレベルのデバッグはやはり必須でしょう。そうなると、市販の開発ツールを利用するということになります。

♦MPW Tools

Tools フォルダの MPW Tools フォルダには、MPW（Macintosh Programmers Workshop）で JDK Tools にある各ツールを利用できるようにするスクリプトなどがあり、MPW Shell で Java のソフトウエア開発を可能にするものを提供しています。MPW Shell はコマンドライン形式でツールを呼び出せるので、MPW を使った方が、他のプラットフォームの JDK による開発により近い感じで作業ができるかもしれません。また、その場合は、MPW Shell 自体で Java のソースコードを作成するということも可能でしょう。

♦その場で Java のプログラムを実行する Java Diddler

Tools フォルダの Other Tools フォルダにある Java Diddler は、入力した Java のプログラムをその場で実行するアプリケーションです。たとえば、1 行だけ Java のプログラムを入力してそれを実行するということができます。あるメソッドがどのような動きになるのかなどを調べるのに利用できなくはありません。ドキュメントによれば、位置付けとしては、Java の学習用に利用できるユーティリティだとされています。

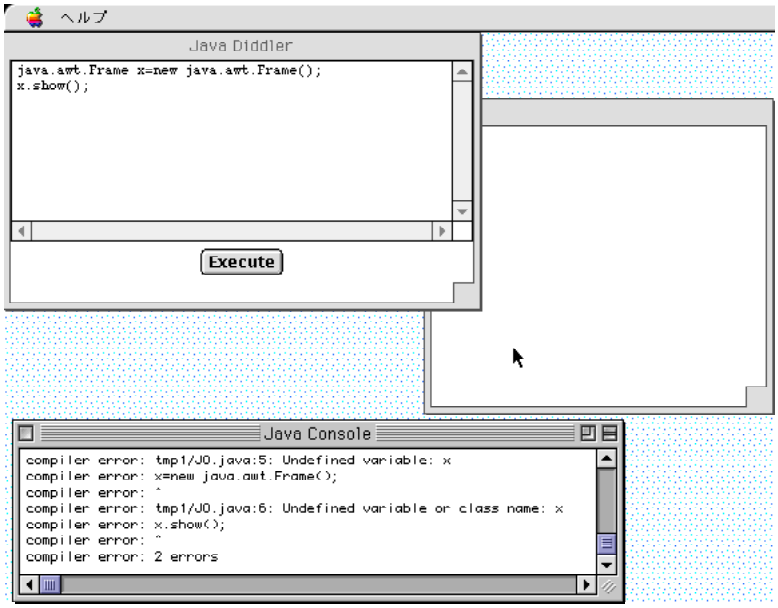

図 1-14　Java Diddler を使った様子。左上のウィンドウでコマンドを入力する。右のウィンドウは実行によって作られたもの。Java コンソールには処理結果が報告される

♦ドラッグ&ドロップでコンパイル

JDK Tools の各アプリケーションは、Finder 上での基本的なドラッグ&ドロップに対応しています。たとえば Java のソースファイルを javac にドラッグ&ドロップすると、そのソースファイルをコンパイルするために開きます。ただし、その場合は、ダイアログボックスで設定を行ってコンパイルの開始をマウスボタンでクリックして指示しなければなりません。

Other Tools の Droplets フォルダにあるサンプルのドロップレットを利用すると、Java のソースファイルをドラッグ&ドロップすれば、javac が起動して自動的にコンパイルが始まり、自動的に javac が終了します。つまり、コンパイル作業が自動化されるというわけです。

JDK Tools にある各ツールは、ファイルを開くといったな基本的な AppleEvent に加え、処理を実行するなど少しは AppleScript に対応しているわけです。それについての解説文書も Droplets フォルダにはありします。

市販の開発ツール

Macintosh 向けに市販の Java 開発ツールも存在します。JDK Tools だけでは、特に

デバッグの作業ができないに等しいため、開発には、開発ツールを購入して臨むことになります。完全にクロスプラットフォームを狙うのであれば、Mac OS の開発ツールにこだわらなくてもいいのですが、Mac OS の機能を使うとなれば、やはり Mac OS 上で開発をするのが適切でしょう。

以下、代表的な開発ツールについて概略を説明しますが、開発ツールに関する使い方などは、Macintosh Java Report では話題に応じて随時説明をしましょう。また、バージョンアップなどのタイミングで詳細に紹介することも検討しています。

表 1-2　開発ツールの比較

ツール	提供元	価格	ビジュアル環境	ソースデバッガ
MRJ SDK	Apple	フリー	×	×
Code Warrior Pro	Metrowerks	¥82,000（アカデミック版は¥28,000）	△	○
Discover Programming for Macintosh	Metrowerks	¥28,000	△	○
Visual Cafe for Java	Symantec	¥36,000（プロフェッショナル版）、¥78,000（データベース開発版）	○	○

♦Code Warrior

C/C++でのアプリケーション開発ツールとしては、Mac OS 環境では事実上 Metrowerks 社の Code Warrior しかなくなってしまったこともあって、Mac OS で開発をしている人は必ず所有している開発ツールではないかと思われます。その Code Warrior でも Java の開発はできます。統合開発環境内で、Java のソースコードを記述して、それをコンパイルし、デバッグをしながらの実行もできます。Java VM は独自のものを提供しており、また、Apple Applet Runner ではなく、独自の Java ソフトウエア実行環境も提供しています。なお、MRJ の VM を使うように切り替えることもできます。

統合開発環境とは別に、Constructor for Java として、ビジュアル開発環境も利用できるようになっています。そこでは、ボタンなどのオブジェクトをウィンドウ上に配置するようなグラフィカルな設計ができ、その設計通りに機能する Java のソースコードを作成します。しかし、そこで生成されたソースを実行するには、Metrowerks が独自に定義したクラスも利用しなければなりません。オブジェクトの 1 つ 1 つを実現するのに標準的なクラスをそのまま使っているのではありません。

なお、初心者向けという位置付けの Discover Programming シリーズも Metrowerks 社からリリースされています。C や C++でのプログラミングでは、PowerPC 対応のコードをコンパイルできないことや、あるいは商用のソフトウエア開発がライセンス上できないことなど制約はありますが、Java プログラミングに関してはあまり大きな制約はなく、Java を中心に利用するのであれば、こちらも検討の価値はあるでしょう。

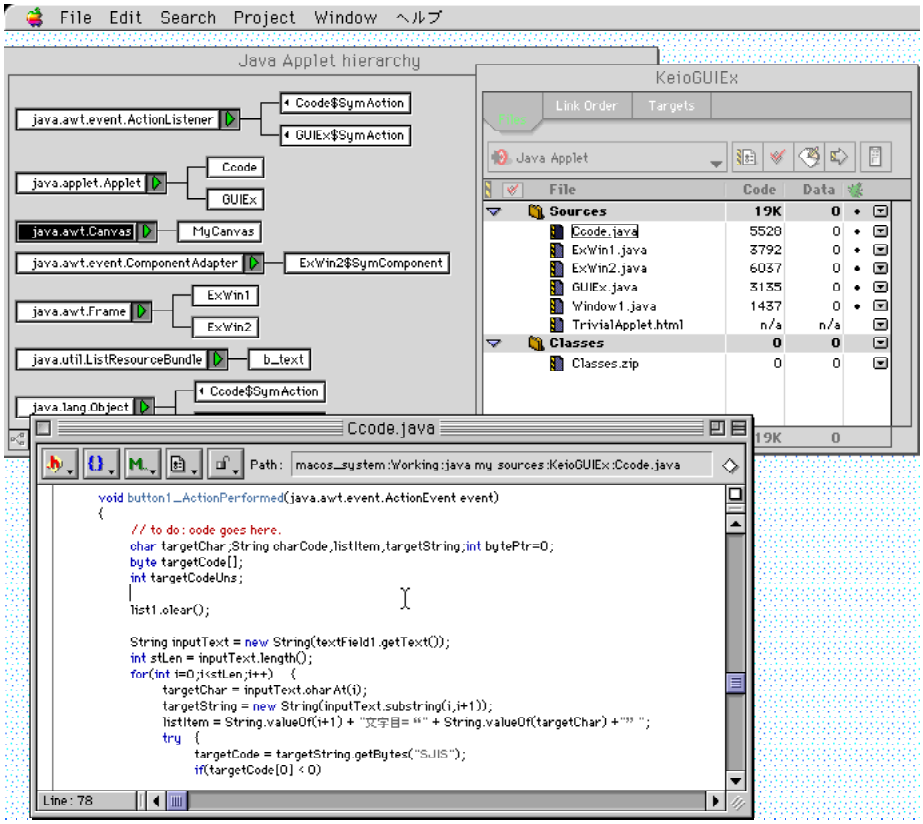

図 1-15　Code Warrior **の作業画面**

♦Visual Cafe

Symantec 社の Visual Cafe for Java は、ビジュアル開発環境が完全に統合された開発ツールです。ウィンドウを用意して、その中にツールを使ってオブジェクトを配置します。そうすると、その設計通りに機能する Java のソースコードが背後で作られます。実行時にそのオブジェクトをクリックして行う作業などを、ソースとして書き加えることができますます。標準的なコンポーネントに加え、独自のコンポーネントも数多くの提供しています。

デバッグや実行は、Apple Applet Runner を利用します。従って、MRJ が実行環境ということになります。「データベース開発版」も存在し、データベースを利用するような Java のアプレットを簡単に作成する機能も利用できます。ただし、データベースエンジン自体は、Windows NT を利用するのが前提になります。

なお、Windows 版は、2.0、2.1、2.5 とバージョンをアップしてきて、Swing にも対応するようになっています。Macintosh 版は、2.0 以降リリースされておらず、バージョンアップされる予定についても全くアナウンスがないという状態です。

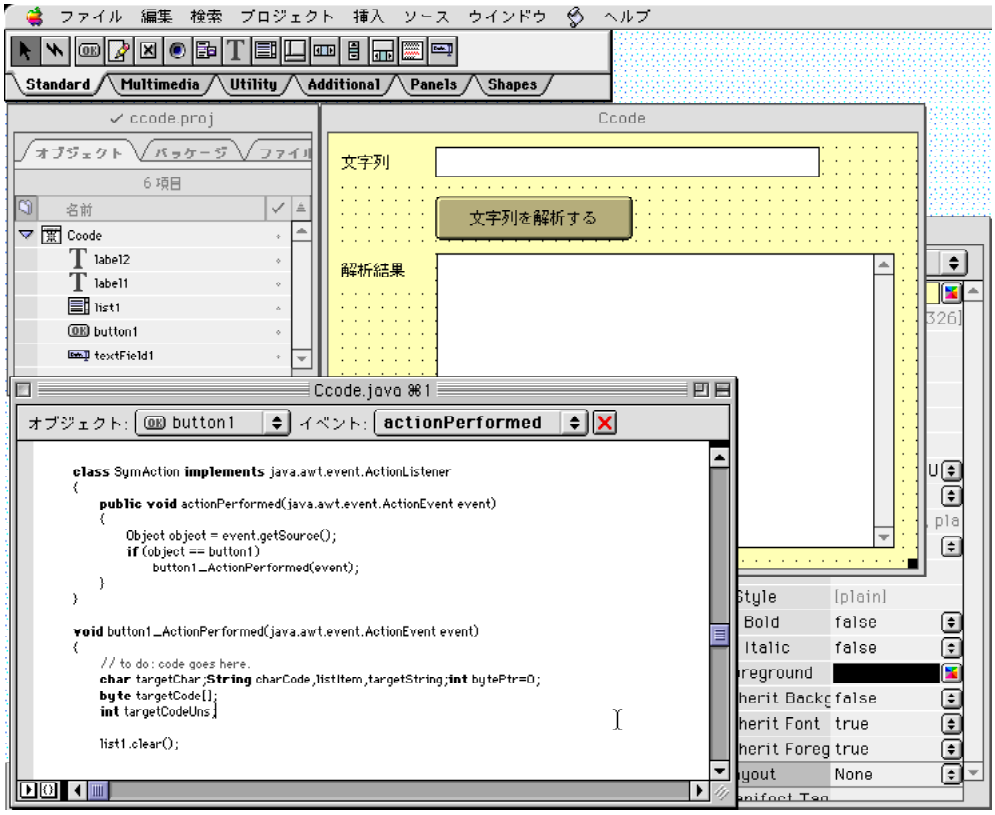

図 1-16　Visual Cafe **の作業画面**

♦どちらの開発ツールがいいか？

非常に難しい問題ですが、Java 開発としてみたとき、Code Warrior と Visual Cafe はやや性質が違うと言えるでしょう。

まず、開発ツール自体の操作性では、Code Warrior の方が良いと考えられます。ソフトウエア自体が軽く動作することや、デバッグ時のキーボードショートカットなど、統合開発環境の使用感は、Code Warrior に軍配が上がるでしょう。

一方、ビジュアル開発環境としてみたときには、Visual Cafe の方がより高機能ですし、何と言ってもフォームエディタなどのビジュアル開発機能が完全に開発環境に統合されているところが使いやすいところです。

Code Warrior は、現在は半年ごとのリリースとなっていますが、そのため、開発途中のバージョンのツールを使うことになってしまうのかもしれません。Code Warrior Pro3 でも、完全な日本語対応はなされておらず、日本語の文字列はそのままソースコードで記述することができません。Code Warrior は、それなりに Java の知識があるプログラマが、コードをばしばし書くような作り方ではそれなりに適応があると考えら

れます。

一方、Visual Cafe の方が初心者には取っ付きやすい面はあるでしょう。ただ、ビジュアル開発ですべてがうまく行くわけではありません。実用的なプログラムにするためには、やはりソースを書く必要があり、場合によってはビジュアル開発環境から離れた作成も必要になります。さらに、Mac 版のバージョンアップがなされていないことも気掛かりな点です。

バグ情報の入手

開発はバグとの戦いです。自分の作ったソフトのバグは仕方ないとしても、実行環境やあるいはシステムのバグについては気が付かないと相当に悩まされることにもなりかねません。Macintosh での Java 開発となると、やはり MRJ 自体のバグについては知っておきたいと思うところでしょう。

Apple では、MRJ についてのバグ情報を Apple の Web ページで公開しています。ときどき、それをチェックする必要もあるでしょう。

また、基本クラスライブラリ自身のバグについては、JavaSoft 社の Web ページで参照できます。JDC（Java Developer Connection）の中で公開されていますが、JDC 自体は入会申し込みが必要です。ただし、無償なので、入会しておいて損はないでしょう。

これらの Web ページのありかについては、この Macintosh Java Report に「リンク情報」としてまとめておきます。

1-6　Mac OS で Java 開発する意義とは

ここまでのところで、Java についての概略を追ってきました。出発点が「マルチプラットフォーム」を目指した Java なのですが、Macintosh 向けに開発することにどのような意味があるのかを検討してみましょう。

Macintosh 環境での Java

まず、Mac OS という十分に完成度の高い OS があるのに、その上にもう 1 つ Java という OS を稼動させるようなことが実際に意味があるのかという素朴な疑問があるでしょう。これについては意味があると考えます。もちろん、C++などと使った Mac OS 向けの開発を否定するわけではありません。C++でネイティブ開発は重要な手段の 1 つです。それに加えて、Java で開発するということも選択肢の 1 つとして浮かび上がらせることができるのではないかと考えられます。

ただし、Java のメリットを生かした開発を行うためには、Mac OS で Java が高いレベルでサポートされている必要があります。幸いなことに、Apple は、Java に対して積極的な対応を行い、MRJ として OS にタイトに組み込んだ実行環境を提供しています。そして、Finder 上でのドラッグ&ドロップを始めとして、OS 機能をふんだんに取り込んだ Java のアプリケーションを作成するということも可能にしています。現状では完全とは言えないものの、少なくとも、Apple は Mac OS 上で実行する Java のソフトウエアは、本来の Mac OS で機能するソフトウエアと遜色なく利用できるということを目指していることは間違いないでしょう。現状では、実行環境のパフォーマンスが悪いとか、Mac OS 独自の機能を利用しづらいなど環境面で整っていないということがありますが、時間が経過すれば、徐々に解決されてくると考えられます。

現在のところ、Mac OS 8.1 が最新版の OS ですが、8.5 のリリース、そして Mac OS X のリリースが計画されており、そのロードマップの中でも Java のサポートは強化される傾向にあると言えるでしょう。MRJ のリリースが OS にうまく連動していない実情はあるものの、Mac OS X のような根底から作り直される OS においても、Java のサポートははっきりと明言されています。Mac OS で Java が使えなくなるということはあり得ない状況ですし、単に使えるということではなく、Mac OS 上で高い機能を発揮するような Java のソフトウエアも実現へ向けて着々と進んでいるということに

なるでしょう。

現状でも、それなりに Mac OS の本来の機能を Java のソフトウエアから利用できます。しかしながら、そのためのドキュメントなど、開発に必要な情報が圧倒的にありません。少ないというよりも、ある部分に関してはまったくないと言ってもいいくらいです。Macintosh Java Report では、こうした情報を提供することによって、Java では貧弱なソフトしか作ることができないという状況を変えることを目指します。

Java で作ったソフトウエア

Web ページに埋め込まれるアプレットは、いろいろなブラウザで機能するため、特定の OS でしか機能しないようにするのは、もちろん得策ではありません。また、それ以前に、いろいろな面でセキュリティがかかっているため、できることもある程度限られます。アプレットはそうした制約を意識した上で、プログラム作成を行うことが必要になります。

一方、アプリケーションソフトは、セキュリティの制約は基本的にはなく、自由にソフトウエアを作成できます。それでも、Java のメリットの 1 つである「マルチプラットフォーム」であることを削がないようにするには、OS に依存しない純粋な Java 機能だけを使うという傾向はあるでしょう。また、Sun は「100% Pure Java」という称号を認定して与えるということも行っています。「完全に Java で作らないといけないということはないのだ」ということは、インタビューなどで強く主張されていますが、「100% Pure Java」という言葉が一人歩きし、「すべての純粋な Java 機能だけでソフトを作ることが最善である」という極端な純血主義的な考え方が広まっているようにも思われます。たとえば、データベースソフトの「ファイルメーカー Pro」の開発版は 100% Pure Java 認定商品です。もちろん、ファイルメーカー Pro 自体は Java で作られているわけではないのですが、Web サーバーとして利用したときに、サーバーのコントロールなどで Java で作られた部品を利用して開発ができます。つまり、Java をベースにしてクロスプラットフォーム対応のシステムを開発できる点が評価されているのです。

♦OS 独自の機能を使うとどうなる？

いずれにしても、100% Pure Java という響きに酔いしれていると、「100%はいい、だけどそれ以外はダメ」という極端なデジタル的結論に走りがちになります。しかし、それではソフトウエアの可能性を削いでいるとしかいいようがありません。

仕様を満たすソフトウエアを作りたい、あるいは要求を満たしたいことから、どうしてもOSの本来の機能を使うしかいないということもあり得ます。たとえば、UNIXやMac OSではエイリアスを利用し、Windowsではショートカットを利用したいという場合があるかもしれません。それらエイリアスやショートカットを作ったり、あるいは参照の解決を行うとなると、Javaの機能だけではできません。たとえば、Mac OSでエイリアスの機能を使う部分を、JDirectあるいはそれをベースにした独自のクラスで利用したとします。そうすると、もちろんその部分は別のOSでは機能しません。結果的には、OSごとに処理を分岐させて対処することになるでしょう。

こうした作りのソフトウエアでも、たとえば95%は共通の標準的なJavaクラスライブラリを使っていて、残り5%はOSごとの処理が組み込まれているというのであれば、Javaのクロスプラットフォーム対応ということは十分に生かされていると言えるでしょう。クロスプラットフォーム対応がなければ、それぞれのOSごとに開発作業をしなければなりません。もちろん、すべてのOSで1からということはないにしても、GUIが絡むようなものだったら、ほとんどOSごとに1からやるのに変わりないでしょう。しかしながら、Javaならば、GUI部分は共通でOSに依存する部分をそれぞれのOSに向けて作るということが可能です。

クロスプラットフォームということを、OSには一切依存しないというところだけに短絡的に結び付けるのではなく、OSに依存したところもあるけども、共通部分をなるべく多く使うような見方に変えて考えてみましょう。そうすると、JavaでMac OSの機能を使うということは、クロスプラットフォームに反することだとは言えません。むしろ、クロスプラットフォーム対応のソフトウエアを作る足掛かりにもなります。OSに依存しない部分だけを使ってソフトウエアを作り、完成度が低くなることよりも、OSに依存する部分を認識しつつ、OSごとの切り分け、そして依存する部分としない部分をうまく切り分けて開発を行うことが、作成するソフトウエアの可能性を高め、より高い完成度が得られると言えるのではないでしょうか。

♦OSの機能を使うのは難しい？

それでは、JDirectのような機能を積極的に使う必要があるのでしょうか。このことはJDirectのところで説明している通り、CとJavaの接点をどうこうするような難しいところは、Appleがクラス定義してくれています。プログラマは、その定義されたクラスを単に使うだけなので、その仕様さえ分かっていれば、Javaの流儀でMac OSの機能は使えます。ただ、その仕様自体、現状ではまったくドキュメント化されていないということが問題になります。

Java のソフトウエアとなると、必ず実行速度の問題が出てきます。いまだに、初期の頃のゆっくりとぎこちない動きしかしない Java アプレットのイメージがあるようなら、認識を変えるべきでしょう。Java で作ったソフトウエアを高速に稼動させる技術は発達し、また Java VM の処理能力も高まっています。また、パソコン自体の高速化もとどまるところを知らないくらいで、パフォーマンスに関する問題は現状ではだいぶんと解決されてきています。

Macintosh で開発を行う理由付けとなる Java

Java がクロスプラットフォームであることは言い換えれば、Mac OS 上で、Windows で機能するソフトウエアを開発できるということに他なりません。開発にもいろいろな形態があるかもしれません。もちろん、市販のソフトウエアの開発のようなものから、個人のツールまでいろいろあります。企業内などのプロジェクトとしてソフトウエアを開発するという場面もあるでしょう。最近では Macintosh のシェアの低下によって、企業の情報システム部門が Macintosh の存在自体を認めないような状況も出てきているという話はあちらこちらで耳にします。たとえば、実験的なシステムであっても、Macintosh で構築することが許されないという具合です。

Mac OS の魅力となると、読者の方々には釈迦に説法でしょうから多くは語りません。やはり何と言われようと、自分は Macintosh を使いたいと思うところかも知れません。そのような状況では、単に Macintosh で開発するという案件は通りづらいのですが、Java を使ってクロスプラットフォーム開発を行うという理由があれば、実作業は Macintosh 上で行うという手法を提示できるでしょう。Windows でも動くということをうまく強調すれば、案件も通りやすいかもしれません。

利用範囲が限定されたような開発だけでなく、市販ソフトやフリーソフトでも、Java で作れば比較的簡単にクロスプラットフォームが実現します。現状でそうしたソフトウエアが少ない理由としては、1 つにはパフォーマンスということがありますが、やはり 100% Pure という言葉に縛られている面があるのではないでしょうか。そのため、要求する機能が実現できず、C や C++で作ったようなソフトウエアに対抗するまでのスペックが実現できていないのです。Java で作るときに、OS に依存する部分もうまく取り込んで高性能なソフトウエアに仕上げることができれば、Java で作ったからというデメリットはかなりは解消されます。そして、少ない労力でクロスプラットフォーム対応が可能になります。

Java で作るということを、Macintosh で開発する理由付けに利用するというのは、ある意味では後ろ向きなことかもしれません。しかしながら、その先にクロスプラットフォームを実現しつつ、最小公倍数ではなく、妥協をしないソフトウエア作りを行うことで、より大きな効果を得られるのではないでしょうか。

見えてこない？Windows の状況

実際にクロスプラットフォームを実現するとなると、どうしても、Windows の OS 機能を使ったプログラムを作成したり、あるいは UNIX の機能を使う必要も出てきます。特に Windows は Java という環境を OS メーカーである Microsoft 社が独自の定義を行い、本来は共通化しているような GUI 処理の部分にまで、Windows 独自の環境を構築しています。そのあたりが、Apple の取り組みと大きな違いです。Apple は GUI などの共通化された部分はその共通化された機能を使い、Mac OS にしかない独自の機能を、Apple 独自のクラスライブラリで利用できるようにしているのです。

Windows での Java の扱いは、Sun と Microsoft との訴訟に発展してしまっているほど簡単には解決しない問題になってしまっています。その意味では、Windows で Java がどうなるかということはまだまだ不確定要素が存在し、スタンスを取りづらい状態ではないでしょうか。もっとも、Microsoft はそうしたところを狙ったとも見えるところです。

Macintosh Java Report は Mac OS における Java 利用をターゲットにしているので、Windows 環境での開発を詳しく取り上げることはしませんが、クロスプラットフォーム開発となると、どうしても Windows の状態を見なければなりません。対岸の事とは思わずに、Windows での状況からも目を離さないようにしなければならないでしょう。

（第 1 章終わり）

LOCUS Report

第2章　MRJ でのシステムプロパティ

- Mac OS では、MRJ（Mac OS Runtime for Java）の環境下で、Java で作成したアプリケーションやアプレットなどを使うのが中心的になるでしょう。実行環境についての情報はシステムプロパティから得られます。
- この章では、MRJ 環境下でのシステムプロパティがどうなっているのかを実際に紹介し、稼動 OS などをプログラム上で知る方法などを説明します。

2-1　実行環境の取得

Java の実行環境において、システムプロパティは、ソフトウエアを稼動している環境についてのさまざまな情報を得る手がかりとなります。まず初めに、システムプロパティについての一般的なことを説明しましょう。

システムプロパティとは

一般にさまざまな設定は、プログラム上では変数に保存し、OS なら OS 独自の方法で保存します。Windows では、環境変数として、OS に何らかの設定を記録し、それをプログラムで参照したりできます（Java からは直接は参照できません）。Mac OS だと、初期設定フォルダにある初期設定ファイルに、たとえばリソースとして保存されたり、あるいはデータとして保存されているかもしれません。いずれにしても、ソフトウエアの動作の上では設定を記録する方法というのは多岐にわたります。

Java の実行環境では、こうした設定の保存方法として「プロパティ」というメカニズムを提供しています。プロパティは、ある設定を名前を指定して参照したり書き込みができるようにしたものです。概念的に、プロパティというデータの記録方法があると考えれば良いでしょう。少し具体的に言うと、プロパティを管理するクラスが定義されています。そして、そのクラスをもとにインスタンスを作成して、プログラム上で独自にプロパティを利用することもできます。1 つのプロパティ管理オブジェクトでは、複数の名前付きプロパティを管理できます。

Java の実行環境では、このプロパティという機能を利用して、稼動しているシステムの状態などをプログラムに伝達することができます。このような実行環境があらかじめ用意しているプロパティを「システムプロパティ」と呼んでいます。Java プログラマから見れば、システムプロパティは、自動的に用意されているプロパティです。一般に書き換えはできませんが、プロパティの値を読み込むことで、たとえば稼動している VM の種類などをプログラム中で特定できます。Mac OS と Windows で異なる動作を組み込むような場合、if 文などで条件分岐が必要ですが、その場合の手がかりになる情報は、システムプロパティから得られるというわけです。

プロパティを管理するクラス

一連のプロパティ情報を管理するために定義されているのは、java.util パッケージ

にある Properties クラスです。このクラスは、abstract クラスの Dictionary、それを extends した HashTable があって、さらに HashTable クラスを extends して Properties クラスが定義されています。

java.util
Disctionary→HashTable→Properties

名前を付けてデータを記録するという機能は、Dictionary クラスで定義されています。HashTable は、名前からデータを取り出すメカニズムをより高速化したものと考えれば良いでしょう。文字通りハッシュテーブルをオブジェクト内に持っていて、名前（キー）からハッシュ値を求め、その値とテーブルを突き合わせて、テーブルから実際のデータを取り出します。つまり、記録している値を調べるのではなく、キーから求められる一定の範囲のデータを調べるような手法がハッシュです。ハッシュ値については、java.lang.Object クラスで定義されている getHash()メソッドで求めますが、このメソッドはネイティブメソッドなので、具体的な計算方法については環境に依存すると思われます。

Properties は、HashTable を、ある意味でシステムプロパティなどに使いやすいように拡張したものです。ストリーム（ファイルやネットワークからの連続データ）から簡単に HashTable、すなわち名前付きの値を保持するオブジェクトを作成したり、あるいはその内容をストリームに書き出す機能が加わっています。また、システムプロパティについても、Properties クラスのオブジェクトへの参照が得られるので、実用的にはこのクラスを利用することになるでしょう。もちろん、HashTable を継承しているので、たとえば、値の書き込みなどについては HashTable で定義されているメソッドを利用することになります。

システムプロパティの取得

システムプロパティを取得するには、以下のようなメソッドを利用します。java.lang.System クラスで定義された Static で定義されたクラスメソッドを利用するので、考え方としては、システムプロパティについては最初から実行環境がどこかにインスタンスを用意してくれていて、それらは以下のメソッドを使えば教えてくれると理解すればよいでしょう。

アプリケーションではシステムプロパティのすべてが取得できますが、アプレットではセキュリティがかけられていて、取り出せないプロパティもあります。したがって、getProperties のようなすべてのプロパティを取得するメソッドは、アプレットで

はセキュリティの例外が発生してしまい、実用上は使えないということになります。

表 2-1　システムプロパティを取得するメソッド

返値の型	メソッド	機能と利用方法
Properties	System.getProperties( )	すべてのシステムプロパティを得る
String	System.getProperty(String key)	引数に指定したキー（名前）のプロパティを得る。結果は文字列で得られるが、キーが存在しないものの場合 null が戻される
String	System.getProperty(String key , String def)	引数に指定したキー（名前）のプロパティを得る。結果は文字列で得られるが、キーが存在しないものの場合引数 def の値が戻される

アプリケーションで getProperties によって取り出したシステムプロパティから、個別のプロパティ値を得たり、あるいは一覧表示するには、以下のようなメソッドを利用します。

表 2-2　プロパティを取り出す Properties クラスのメソッド

返値の型	メソッド	機能と利用方法
String	getProperty(String key)	引数に指定したキー（名前）のプロパティを得る。結果は文字列で得られるが、キーが存在しないものの場合 null が戻される
String	getProperty(String key , String def)	引数に指定したキー（名前）のプロパティを得る。結果は文字列で得られるが、キーが存在しないものの場合引数 def の値が戻される
void	list(PrintStream out)	プロパティとして定義されている名前と値をすべて引数のストリームに書き出す

一般にはプログラム中ではシステムプロパティに独自にプロパティを追加することは行いません。独自にプロパティを管理したいときには、Properties クラスのインスタンスを別途作成して、そのインスタンスに追加を行います。追加を行うには、HashTable クラスに定義されたメソッドを使いますので、詳細は HashTable クラスのリファレンスを御覧ください。

2-2 プロパティの取得結果

実際に MRJ 環境でプログラムを実行して、プロパティを取得してみましょう。取得するプログラムについての概要と、取得結果をここで説明します。

取得するプログラム

システムプロパティを取得するプログラムとして、アプリケーション用とアプレット用を用意しました。プログラムのソースコード自体は、この章の最後の 2.4 節にまとめました。

アプリケーションもアプレットも、PropSysConsole クラスを利用してシステムコンソールにプロパティ値を表示します。まず、このプログラムから説明しておきましょう。

PropSysConsole は、必要な作業をコンストラクタの中で全て行ってしまいます。単にオブジェクトのインスタンス化のときに、システムプロパティを読み出してコンソールに表示しているだけです。

最初に、アプリケーションで実行させて、GetProperties と list メソッドを使い、システムのプロパティを全部表示させてみました。すると、相当たくさんのプロパティがあるのですが、多くはフォント設定のためのプロパティのようです。この章ではフォントについては詳細はチェックしないことにします。システムプロパティの一覧から有用なものなどを取り出してコンソールに表示するように作ってあります。System.getProperty によって特定のプロパティを取り出しますが、引数には決められたキーワードを指定しなければなりません。キーワードと取り出し結果については、この後に一覧表で示しましょう。

なお、アプレットではセキュリティがかかっていて取りだせないプロパティがあるので、PropSysConsole のコンストラクタに、現在アプレットで実行しているかどうかを指定する引数を定義してあります。

GetCodeString メソッドは、得られる文字列の内容をコードで表示するための一種のサブルーチンです。システムプロパティで改行コードが得られますが、それは改行コードそのものの文字列です。その文字列をコードでコンソールに表示するためにこうしたメソッドを用意しました。

♦アプリケーションでの実行

GetSysPropApp.java がアプリケーションとして最初に実行されるクラスです。つまり、ここに main メソッドを定義していますが、単に PropSysConsole クラスを new で作成して、クラスのコンストラクタを呼び出しているだけです。

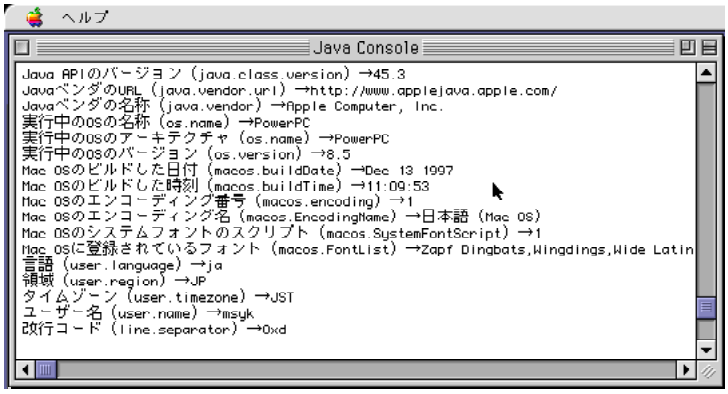

図 2-1　　アプリケーションを実行してシステムプロパティを表示

♦アプレットでの実行

実行を開始させるためのアプレットとして、GetSysProp.java を作成しました。やはりアプレット起動時に、PropSysConsole クラスを new で作成して、クラスのコンストラクタを呼び出しているだけです。Applet Runner で実行した時には、自動的にコンソールが表示されますが、ブラウザで起動したときにはコンソールはメニューから選択しないと表示されません。そこで、アプレットにはその旨をメッセージとして表示するようにしました。Visual Cafe を使ってアプレットを開発しているので、Visual Cafe が独自の処理に使うコメントなどがありますが、単に、java.awt.Label を利用してアプレット上に文字を表示しているだけです。

なお、PropSysConsole クラスのコンストラクタでは、アプレットとして実行したときに取り出せないプロパティについては、if 文で取得をしないようにプログラムされています。アプレットからコンストラクタを呼び出す時に引数には true を指定します。取得できるプロパティの種類も少ないので、図に示すようにプロパティの項目も少なくなっています。

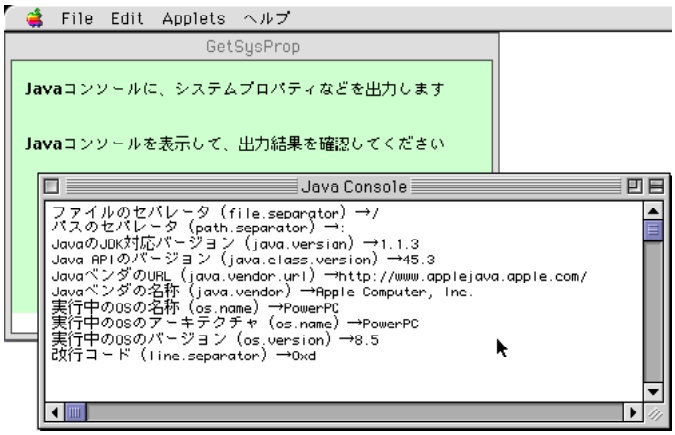

図 2-2　Applet Runner **で実行した結果**

プロパティの取得結果

実際に取り出した結果を表にしました。実行環境は以下の通りです。

- **Mac OS 8.5（日本語版）**
- **MRJ 2.0（Mac OS 8.5 に付属）**
- **Power Macintosh 7600/200**

表 2-3　**取得したプロパティの一例**

取得できる内容	GetProperty の引数	アプレット	取得値
ファイルエンコーダのパッケージ	file.encoding.pkg	×	sun.io
ファイルエンコーダ	file.encoding	×	MacTEC
ファイルのセパレータ	file.separator	○	/
パスのセパレータ	path.separator	○	:
クラスへのパス。システムクラスへのパス	java.class.path	×	（別掲）
ワーキングディレクトリ。実行ファイルのパス	user.dir	×	/macos_system/Working/MJR%20Samples/ch02_SysProp
Java VM のディレクトリ	java.home	×	/macos_system/%83V%83X%83e%83%80%83t%83H%83%8b%83_/%8b%40%94%5c%8ag%92%a3/MRJ%20Libraries
ユーザーのワーキングディレクトリ	usr.home	×	null
Java の実行環境	java.compiler	×	MRJ2Jitc
Java の JDK 対応バージョン	java.version	○	1.1.3

取得できる内容	GetProperty の引数	アプレット	取得値
Java API のバージョン	java.class.version	○	45.3
Java ベンダの URL	java.vendor.url	○	http://www.applejava.apple.com/
Java ベンダの名称	java.vendor	○	Apple Computer, Inc.
実行中の OS の名称	os.name	○	Mac OS
実行中の OS のアーキテクチャ	os.arch	○	PowerPC
実行中の OS のバージョン	os.version	○	8.5
Mac OS のビルドした日付	macos.buildDate	×	Dec 13 1997
Mac OS のビルドした時刻	macos.buildTime	×	11:09:53
Mac OS のエンコーディング番号	macos.encoding	×	1
Mac OS のエンコーディング名	macos.EncodingName	×	日本語（Mac OS）
Mac OS のシステムフォントのスクリプト	macos.SystemFontScript	×	1
Mac OS に登録されているフォント	macos.FontList	×	Zapf Dingbats,Wingdings,Wide Latin,Webdings,Verdana,Trebuchet MS,Times New Roman,Times,Textile,Techno,Tahoma,Symbol,Stencil,Sand,（以下省略）
言語	user.language	×	ja
領域	user.region	×	JP
タイムゾーン	user.timezone	×	JST
ユーザー名	user.name	×	msyk
改行コード	line.separator	○	0xd（実際にはこのコードだけの 1 バイト長の文字列）

♦java.class.path の内容

/macos_system/%83V%83X%83e%83%80%83t%83H%83%8b%83_/%8b%40%94%5c%8ag%92%a3/MRJ%20Libraries/MRJClasses/JDKClasses.zip:/macos_system/%83V%83X%83e%83%80%83t%83H%83%8b%83_/%8b%40%94%5c%8ag%92%a3/MRJ%20Libraries/MRJClasses/MRJClasses.zip:/macos_system/%83V%83X%83e%83%80%83t%83H%83%8b%83_/%8b%40%94%5c%8ag%92%a3/MRJ%20Libraries/MRJClasses/Properties.zip:/macos_system/%83V%83X%83e%83%80%83t%83H%83%8b%83_/%8b%40%94%5c%8ag%92%a3/MRJ%20Libraries/MRJClasses/Symantec%20Debug%20Classes.zip:/macos_system/%83V%83X%83e%83%80%83t%83H%83%8b%83_/%8b%40%94%5c%8ag%92%a3/MRJ%20Libraries/MRJClasses/:/$VFS

2-3 システムプロパティの利用

システムプロパティで実際に得られる値についてここでは検討してみましょう。必要に応じてシステムプロパティの値を利用すればいいのですが、代表的なものについて解説をしておきます。

実行 OS の判断

実行 OS の判断には、「os.name」を利用するのが順当なところでしょう。ただし、実行 OS が PowerPC の場合は「PowerPC」という文字列が得られ、CPU が 68k では「MC68k」という文字列が得られます。つまり、Mac OS でも CPU によって os.name で得られるプロパティは変わってくるということになります。Mac OS 上で機能しているかどうかは、これらのいずれかのプロパティであることを判定することになるでしょう。たとえば、次のようになります。

```
String osName = System.getProperty("os.name");
if((osName.equals ("PowerPC")) || (osName.equals ("MC68k")))    {
    /*      Mac OS の場合の処理          */
}
else      {
    /*      Mac OS でない場合の処理          */
}
```

なお、Microsoft の VM を使っている場合でも、PowerPC という文字列が得られます。

java.vendor で得られる値に Apple が含まれていれば Mac OS という考え方もできるのですが、Internet Explorer を使っていて Microsoft の VM を使っていれば、「Microsoft Corp.」という文字列が得られます。むしろ java.vendor は、アプレットで実行している VM の種類を特定するために利用できるということになるでしょう。たとえば、次のようにして VM ごとに処理を切り分けます。もちろん、Mac OS で稼動していることが前提になります。

```
String vendorName = System.getProperty("java.vendor");
if(vendorName.indexOf("Apple") >= 0 )       {
    /*      MRJ の場合の処理              */
}
else if(vendorName.indexOf("Microsoft") >= 0 )    {
    /*      Microsoft の VM の場合の処理          */
}
else {
    /*      上記以外の VM の場合の処理            */
}
```

¥アプリケーションでは、Mac OS かどうかを判断すれば、MRJ かどうかは分かります。ブラウザ外で稼動する Mac OS 用の VM は基本的には MRJ 以外には存在しないと言えるからです。

Mac OS のバージョンについては、os.version によって得られるプロパティ値を使えばすぐに判断できます。得られるのは文字列であって、数値ではありませんので、注意が必要です。

実行環境についての情報

場合によっては、JDK のバージョンに応じて機能を分岐したり、あるいは使えない機能を設定する必要があるかも知れません。その場合には、java.version を利用して得られたプロパティ値を使えばよいでしょう。

♦実行 OS の言語についての情報

Mac OS で実行しているときには、プロパティの名前として、macos で始まるものがいくつか利用できます。これらのプロパティは、Windows など別の実行環境では使えないもので、つまり MRJ でないと利用できないシステムプロパティと考えればよいでしょう。これらの情報を利用すると、日本語システムなのかどうかなども判断できる模様です。

まず、macos.EncodingName を利用して得たプロパティ値は、OS の言語を示す文字列が得られます。ただ、このプロパティはいわば表示用と考えて、プログラム中でシステムの言語を判断するには、macos.encoding を利用すべきでしょう。この値は、Toolbox の Script Manager で定義されたスクリプトコードだと思われます（スクリプトコードは、0 がローマン、1 が日本、2 が中国、3 が韓国など)。フォントスクリプトを取得する macos.SystemFontScript があるので、こちらが Script Manager の API である FontScript( )を利用し、macos.encoding は IntlScript( )を利用しているものだと思われます。いずれにしても、日本語システム上ではどちらも 1 が得られます。

なお、アプリケーションであれば、user.language と user.region によって、デフォル

トのロケールの言語と領域を知ることもできるようです。また、user.timezone も合わせて、国際化パッケージで定義されている地域に特有の情報も取得できます。

なお、システムにセットアップされているフォントについても、macos.FontList で得られます。ここではカンマで区切られて Mac OS システムでのフォント名が得られますが、MRJ 2.0 だと日本語のフォント名については文字列が UNICODE 化されず、おそらくシフト JIS コードのままになっているので、コンソール上では文字化けして見えます。アプリケーションでフォントメニューを作る時などにはこうしたプロパティを利用することも考えられますが、取得した段階でフォント名が化けてしまっているのは少々やっかいでしょう。マルチ OS を目指した Java なのですが、フォント環境についてはまだまだ取扱いが難しい面があるということです。

♦ファイル関連のシステムプロパティ

ファイル関連のシステムプロパティでは、file.separator によって、区切り文字を取得できますが、一般には Java のプログラム中では MRJ ベースで稼動させるにしてもパスはスラッシュ（/）で区切って指定することが多いでしょう。Mac OS の本来の区切り記号はコロン（：）ですが、MRJ で稼動させるソフトウエアでも Java でのプログラム中ではスラッシュをフォルダの区切り文字として指定します。ただし、Windows 環境では、「¥」を区切り記号に使うこともあるので、マルチプラットフォーム化を完全に行うには、やはり、file.separator で取得した文字列をフォルダの区切り文字列に使用するようにしておく必要が出てくるかも知れません。

path.separator は、java.class.path などのパスが複数含まれているような設定の中で、各パスの記述を区切るための記号です。クラスパスなどのチェックを独自に行うようなときには、この記号を手がかりに、java.class.path で得られた値を区切る必要があるでしょう。なお、こうした作業は java.util.StringTokenizer クラスを使えば比較的簡単です。

アプリケーションが存在するのと同じディレクトリを user.dir で取得することができます。同じディレクトリにあるファイルを開くような場合には、このプロパティ値が手がかりになります。なお、user.home については MRJ での実行中には null 値が戻るため、ユーザーのワーキングディレクトリは取得できません。もともと、OS 自体にユーザーのワーキングディレクトリという考え方がないので、これは仕方ないのかもしれません。

2-4　サンプルプログラムのソース

この章ではシステムプロパティの値を検討するのが主な目的ですが、そのために、プロパティを取得するサンプルプログラムを作成しました。次のように 3 つのソースファイルで構成しています。

SysProConsole.java

```
import java.lang.*;
import java.util.*;

public class SysProConsole extends java.lang.Object
{
    public SysProConsole(boolean isApplet)
    {
//          Properties sysProps = System.getProperties();
//          System.list(System.out);
                //すべてのプロパティをコンソールに出力する

            if(!isApplet){
                System.out.println("ファイルエンコーダのパッケージ（file.encoding.pkg）→"
                                + System.getProperty("file.encoding.pkg"));
                System.out.println("ファイルエンコーダ（file.encoding）→"
                                + System.getProperty("file.encoding"));
            }
            System.out.println("ファイルのセパレータ（file.separator）→"
                                + System.getProperty("file.separator"));
            System.out.println("パスのセパレータ（path.separator） "
                                + System.getProperty("path.separator"));

            if(!isApplet){
                System.out.println(
                                "クラスへのパス。システムクラスへのパス（java.class.path）→"
                                + System.getProperty("java.class.path"));
                System.out.println("ワーキングディレクトリ。実行ファイルのパス（user.dir）
    "
                                + System.getProperty("user.dir"));
                System.out.println("Java VM のディレクトリ（java.home）→"
                                + System.getProperty("java.home"));
                System.out.println("ユーザーのワーキングディレクトリ（usr.home）→"
                                    + System.getProperty("usr.home"));
            }

            if(!isApplet)
                System.out.println("Java の実行環境（java.compiler）→"
                                    + System.getProperty("java.compiler"));
            System.out.println("Java の JDK 対応バージョン（java.version）→"
```

```
                                    + System.getProperty("java.version"));
        System.out.println("Java API のバージョン（java.class.version）→"
                                    + System.getProperty("java.class.version"));
        System.out.println("Java ベンダの URL（java.vendor.url）→"
                                    + System.getProperty("java.vendor.url"));
        System.out.println("Java ベンダの名称（java.vendor）→"
                                    + System.getProperty("java.vendor"));

        System.out.println("実行中の OS の名称（os.name）→"
                                    + System.getProperty("os.arch"));
        System.out.println("実行中の OS のアーキテクチャ（os.name）→"
                                    + System.getProperty("os.arch"));
        System.out.println("実行中の OS のバージョン（os.version）→"
                                    + System.getProperty("os.version"));
        if(!isApplet){
            System.out.println("Mac OS のビルドした日付（macos.buildDate）→"
                                    + System.getProperty("macos.buildDate"));
            System.out.println("Mac OS のビルドした時刻（macos.buildTime）→"
                                    + System.getProperty("macos.buildTime"));
            System.out.println("Mac OS のエンコーディング番号（macos.encoding）→"
                                    + System.getProperty("macos.encoding"));
            System.out.println("Mac OS のエンコーディング名（macos.EncodingName）→"
                                    + System.getProperty("macos.EncodingName"));
            System.out.println(
                  "Mac OS のシステムフォントのスクリプト（macos.SystemFontScript）→"
                                    + System.getProperty("macos.SystemFontScript"));
            System.out.println("Mac OS に登録されているフォント（macos.FontList）→"
                                    + System.getProperty("macos.FontList"));
        }
        if(!isApplet){
            System.out.println("言語（user.language）→"
                                    + System.getProperty("user.language"));
            System.out.println("領域（user.region）→"
                                    + System.getProperty("user.region"));
            System.out.println("タイムゾーン（user.timezone）→"
                                    + System.getProperty("user.timezone"));
            System.out.println("ユーザー名（user.name）→"
                                    + System.getProperty("user.name"));
        }

        System.out.println("改行コード（line.separator）→"
                                    + getCodeString(System.getProperty("line.separator")));

    }

    String getCodeString(String text)
    //引数の文字列をシフト JIS コードを示す 16 進数表示の文字列を求めて返す
    {
        byte targetCode[];
        int targetCodeUns;
        StringBuffer codeString = new StringBuffer("");
        int stLen = text.length();
        for(int i=0;i<stLen;i++) {      //引数の文字列内の各文字について、コード文字列を得る
            try    {
```

```
                    targetCode = text.substring(i,i+1).getBytes("SJIS");
                                        //1 文字をシフト JIS のバイト列に変換
                    if(targetCode[0] < 0)    //1 バイト目が負の数なら、128 以上のコード
                        targetCodeUns = 256 + targetCode[0];   //本来のコードにかえる
                    else
                        targetCodeUns = targetCode[0];    //128 より小さい場合はそのまま
                    codeString.append("0x");     //16 進数を示す文字列をまず用意
                    codeString.append(String.valueOf(Integer.toString(targetCodeUns,16)));
                        //1 バイト目を 16 進コード表記の文字列として得て、
                        //結果の文字列につなげる
                    if(targetCode.length>1)       {       //2 バイト目があるのなら
                        if(targetCode[1] < 0)             //コードが 128 以上の時の処理
                            targetCodeUns = 256 + targetCode[1];
                        else
                            targetCodeUns = targetCode[1];
                        codeString.append(
                            String.valueOf(Integer.toString(targetCodeUns,16)));
                                //2 バイト目のコードを示す文字列を結果につなげる
                    }
                }
                catch(Exception e)     {       System.out.println(e.getMessage());             }
            }
            return codeString.toString(); //結果を戻す
        }
    }
```

GetSysPropApp.java

```
import java.awt.*;

public class GetSysPropApp
{
    static public void main(String args[])
    {
        new SysProConsole(false);
    }
}
```

GetSysProp.java

```
import java.awt.*;
import java.applet.*;

public class GetSysProp extends Applet
{
    public void init()
    {
        //{{INIT_CONTROLS
        setLayout(null);
        setSize(360,189);
```

```
            setBackground(new Color(-3342388));
            label2 = new java.awt.Label(
                        "Java コンソールを表示して、出力結果を確認してください");
            label2.setBounds(9,50,335,24);
            add(label2);
            label1 = new java.awt.Label(
                        "Java コンソールに、システムプロパティなどを出力します");
            label1.setBounds(9,10,342,21);
            add(label1);
            //}}
            new SysProConsole(true);
    }

    //{{DECLARE_CONTROLS
    java.awt.Label label2;
    java.awt.Label label1;
    //}}
}
```

（第 2 章終わり）

第3章　Mac OS 向けのアプリケーション

- Java 言語を利用してアプリケーションを作成することができます。まず、一般的な Java でのアプリケーション作成と、それを Mac OS で実行させる方法の概要について説明します。
- Mac OS でのアプリケーションでは、Finder 上でアプリケーションのアイコンにドラッグ&ドロップ、あるいは文書をダブルクリックして起動するという動作が要求されます。
- 文書を開くだけでなく、印刷についても同様に Finder からの要求を受け付ける必要があります。さらに、終了の要求もあります。
- Java のアプリケーションでもこうした Mac OS でのアプリケーションと同等な動きをさせることができます。

3-1 Java のアプリケーション

Java のアプリケーションの基本的なことと、Mac OS での作成や実行方法の概要を説明しましょう。実際に、開発ツールを使って作成する方法については、第 4 章で説明をします。

アプリケーションのプログラム

ユーザーが起動するソフトウエアのもっとも一般的な形式が「アプリケーション」と言えるでしょう。自動的に起動されることもありますが、一般にはユーザーがダブルクリックなどの操作を行って、アプリケーションソフトを起動することができるように、OS は作られています。OS の動作としては実行ソフトウエアをロードし、そちらに制御を渡すような複雑な動作があるのですが、ユーザーはそのようなことを意識せずに使うことができます。

また、アプリケーションを開発する場合、プログラムがあって、それを特定の OS 上で実行可能にするためには、OS の規則に従ったファイルに構成しなければなりません。ただし、一般にはそうした複雑なことは開発ツール側ですべて自動的にやってくれるのが一般的なので、単に作成するファイル名を指定すると、アプリケーションを作ってくれるというのが一般的でしょう。

♦Java でのアプリケーション

Java のアプリケーションは、Java の VM によってアプリケーションとして認識されるように作っておく必要があります。

まず、Java のプログラムには、Java 言語で作成されたソースコードがあります（.java という拡張子のファイル名となります）。このソースコードをコンパイルすると、JavaVM で実行可能なバイナリファイルが作られます（.class という拡張子のファイル名となります）。その中身は「バイトコード」と呼ばれるバイナリデータです。ソースコード中の１つのクラス定義が、１つの.class ファイルに対応するのが基本です。

この.class の拡張子を持つファイルは、Java VM で実行可能ですが、「実行せよ」という指示を出した時、どのように作成した.class ファイルに制御が移るのでしょうか。それは、実行するクラスの中に定義されてある、以下のようなメソッドを実行することによって実現します。

```
static public void main( String   arg[] ) {      }
```

つまり、最初に main が呼び出されるということが決まりになっているのです。返値の形式や static メソッド、つまりクラスメソッドである点、そしてスコープが public であることは、仕様として決められているわけです。

また、実行する時に、コマンドラインとして与えたパラメータなどは、main メソッドの引数 arg に文字列の配列として得られます。Windows や UNIX で Java アプリケーションを実行する時には、コマンドラインを使うことも多いかも知れませんが、Mac OS ではコマンドラインを指定できない状況が多いため、main の引数の利用はあまり積極的にはできないというところでしょう。

いずれにしても、Java のソフトウエアなので、アプリケーションの場合もまずはクラスを定義します。そしてコンパイルして.class ファイルを作成します。そのコンパイル結果の class ファイルを実行するのですが、そのときに、クラス定義中にある main メソッドを呼び出すという規則になっています。

♦簡単なアプリケーション

たとえば、単にウインドウを表示するだけのアプリケーションとなると、たとえば次のようなプログラムになります。

```
import java.awt.*;

public class MyWindow extends Frame
{
    //コンストラクタの定義
    public MyWindow()
    {
        setVisible(true);          //ウインドウを表示する
        setTitle("Simple Window");   //ウインドウのタイトルを設定
        setSize(200,100);        //ウインドウのサイズを指定する
    }

    //アプリケーションとして実行した時に最初に呼び出されるメソッド

    static public void main(String args[])
    {
        new MyWindow();        //MyWindow のインスタンスを生成する
    }
}
```

このソースプログラムのファイル名は、MyWindow.java です。コンパイルすると、MyWindow.class ファイルが生成されます。そして、なんらかの方法で、この MyWindow.class をアプリケーションとして実行したとします。すると、Java VM はこ

のバイナリファイルをロードして実行可能な状態にし、main メソッドの呼び出しを行うということです。

プログラム自体は、AWT の機能を使ったシンプルなものですが、解説をしておきましょう。まず、Frame クラスを拡張して MyWindow クラスを定義しているので、MyWindow はウインドウとして機能するクラスです。そして、new によって MyWindow のインスタンスが生成されるとき、コンストラクタが呼び出されます。まず、Frame のサブクラスなので、MyWindow を new で生成した段階で、ウインドウ自体がすでに用意されることになります。ウインドウは最初は非表示になっているのでそれを表示し、タイトルやサイズを設定するメソッドを利用しています。アプリケーションとして実行するので、main メソッドが呼び出されますが、そこでは単に MyWindow のインスタンスを生成するということだけを行っています。

アプリケーションを実行する

具体的に、アプリケーションとして作成された.class ファイルを実行する場合、JDK では「java」というそのものずばりの名前のツールを利用できるようになっています。たとえば、上記のようなプログラムでは、

```
java MyWindow
```

のようにすれば実行できるということになります。また、java ツールのコマンドライン中に、実行クラス名以降のパラメータは、Java のプログラム中の main メソッドへの引数として引き渡されます。

♦Mac OS での Java アプリケーション実行

ただし、Mac OS の環境では、java というツールは提供されていません。そのため、現実にはいくつかの方法を取ることになります。なお、具体的な使い方については、第 4 章で手順を示して解説することにします。

まず、MRJ という基本的な Java 実行環境によって提供されている方法としては、JBindery というツールを使うことです。これにより、.class 形式のアプリケーションプログラムを、Mac OS で実行することができます。JBindery は Mac OS のアプリケーションだけに、Finder 上で.class ファイルを JBindery にドラッグ&ドロップすれば、実行することができます。JBindery のこうした使い方は、JDK での java ツールに対応したものとなっています。

JBindery 上で実行すると、JBindery のアプリケーション内で、Java のアプリケーシ

ョンが起動します。そのため、アプリケーションメニューに項目が増えるわけではなく、JBindery というアプリケーション内部での実行ということになります。とりあえず機能させる場合には有効かもしれませんが、実用に使うアプリケーションとしては、次に説明するように、独立した Mac OS のアプリケーションファイルを生成して、それを実行することになるでしょう。

♦アプリケーション形式のファイル

開発ツールによって Mac OS のアプリケーションを生成し、それをダブルクリックなどで起動することで実行できます。また、アプリケーションとして制作している途中にデバッグを行うこともあるでしょうが、そのときには仮のアプリケーションなどを作成することもあります。

ただし、Mac OS のアプリケーションとは言っても、コンパイラによって生成された Java のバイナリ（バイトコード）を含む.class ファイルの中身を持っており、その内容に加え、OS から見れば実際にアプリケーションとしての体をなすような形式にしたものです。Mac OS 環境での Java アプリケーションは、ひじょうに割り切った見方をすれば、.class ファイルの内容を持ち、ファイルタイプが APPL になっているものと見ればよいでしょう。

♦Visual Cafe で作られるアプリケーションファイル

詳細なところは公開されていないのですが、ツールなどで見る限りは、Visual Cafe で作られる Mac OS での Java アプリケーションは次のような構成になっているものと思われます。まず、コンパイル結果の Java バイナリ、すなわち.class ファイルの中身は、1 つあるいは複数のファイル分がアプリケーションソフトのファイルのデータフォークに入れられているようです。

そして、アプリケーションファイルのリソースフォークには、CODE リソースがありますが、cfrg リソースはありません。つまり、Mac OS から見れば、68k アプリケーションだと認識して、起動すると CODE リソースの実行に入るわけです。この CODE リソースでは、おそらく、MRJ の実行環境を整えて、MRJ に実行を移すだけのことをやっているのだと思われます。単純に言えば、Java VM としての MRJ を起動するということになるでしょうか。CODE リソースではありますが、Java の実行環境、すなわち MRJ は PowerPC ネイティブですので、Java プログラム自体は 68k のエミュレーションで実行されているわけではありません。もっとも、Java の VM はエミュレートしていると言えるでしょう。

また、Java アプリケーションには、SIZE リソースを定義して、アプリケーション

のふるまいを指定することも行われています。こうした、Mac OS のアプリケーションとしてふるまうための必要なリソースは一通りファイルに入っていますし、Java の実行オブジェクト（バイトコード）もアプリケーションのファイルに含まれます。そのため、別のパソコンに持ち込む場合には、そのアプリケーションファイルだけをコピーすればいいので、取り扱いも便利です。

ただし、BNDL リソースや関連するアイコンのリソースなどについては、Visual Cafe では生成できません。別途 ResEdit で作成したものを、アプリケーションのビルドのときにマージする機能を利用します。

♦JBindery で作るアプリケーション

JBindery でもアプリケーションは作成できますが、既定値ではそのアプリケーションには、.class ファイルの中身自体は保存されていません。その代わりに、実行するクラスが存在するフォルダへのエイリアスが定義されています。したがって、作成したアプリケーションと、.class ファイルとはセットにしておかないと実行できないわけです。

なお、クラスファイルやフォルダを、アプリケーション自体と相対ディレクトリで指定したり、あるいは VFS（Virtual File System）の機能により、実行に必要なクラスファイルなどを単独のアプリケーションファイルとしてまとめる事も可能です。

JBindery で作ったアプリケーションにも、CODE リソースがあり、これによって MRJ の環境がセットアップされて、Java VM を機能させる処理が含まれているものと思われます。また、SIZE リソースも作られます。

その他のリソースは、ResEdit で作ったものをマージすることができるようになっています。JBindery では、バイトコードが別ファイルになっている以外は、基本的には Visual Cafe に似た構成になっていると言えるでしょう。

3-2 ドラッグ&ドロップを可能にする

Mac OS の Finder では、文書ファイルをアプリケーションにドラッグ&ドロップすることによって、文書を開いたりあるいは処理をすることができます。また、文書ファイルをダブルクリックして開くのも、基本的には同じメカニズムが働いています。Mac OS でのこうした機能の基本的なことと、その機能を持ったアプリケーションを Java で実現するための方法を説明しましょう。

Finder でのドラッグ&ドロップのメカニズム

Mac OS では、AppleEvent によって、アプリケーションあるいは一般的にソフトウエア同士でのやりとりができるようになっています。あるアプリケーションで、別のアプリケーションにさせたい作業をスクリプトとして手順書きを作ることで、そのスクリプト通りに処理をさせることもできます。こうしたスクリプトでの利用は AppleScript として知られていますが、ベースには AppleEvent の機能を使っています。AppleEvent はシステムの機能ですが、プロトコルに近いものと考えれば良いでしょう。アプリケーション間で柔軟に処理ができるように、さまざまな取り決めがなされています。ただし、一般には、AppleEvent に対応するようにアプリケーションを作り込まないといけません。AppleEvent はある意味では別のアプリケーションから送り届けられる「命令」です。命令も 1 種類ということではなくたくさんあるのであれば、それに対応するように、命令ごとに処理を記述する必要が出てきます。

AppleEvent は目に見える部分ではなく、また、アプリケーション間通信と一言で片付けることもできるのですが、完全なレベルを目指すとアプリケーションへの実装にはかなりの労力はかかります。そこで、アプリケーションとして最低限備えていないといけない Required AppleEvent と呼ばれる 4 つのイベント命令が定義されています。それらは、「新規文書作成（New）」「文書を開く（Open）」「文書を印刷する（Print）」「終了する（Quit）」という処理で、いずれも、Finder からの操作に対応するというのがいちばんの趣旨です。

♦Finder でのアプリケーションの起動

4 つの Required Event のうち、Open イベントが非常に重要です。一般には Open イ

ベントに附随して、存在するファイルが 1 つあるいは複数送り届けられます。Open イベントを受け取ったアプリケーションは、その附随した情報にあるファイルを開き、文書ファイルとしてウインドウに表示したり、あるいは処理にかけるように作られている必要があります。

Finder では、アプリケーションをダブルクリックして起動するだけではなく、アプリケーションが作成した文書ファイルをダブルクリックしたり、あるいはアプリケーションへの文書ファイルのドラッグ&ドロップによっても起動できます。Mac OS では、他にもアップルメニューやローンチャーなどの起動方法もあるのですが、背後ではダブルクリックなどをしているのと同じことになっています。

文書ファイルをダブルクリックしたとき、そのファイルのクリエイター情報を手がかりにアプリケーションを起動します。そして、起動したアプリケーションに対して、文書ファイルを開くことを指示する Open イベントが Finder から送り届けられます。アプリケーションが Open イベントに対応していれば、「文書ファイルをダブルクリックして開く」という操作環境が実現すると言うわけです。

もし、文書ファイルに関連づけられているアプリケーションがすでに起動していれば、そのアプリケーションをアクティブにして、やはり Open イベントが送り届けられ、それを受け取ったアプリケーションが文書ファイルを開くという動作を行います。

Finder 上で、文書ファイルをアプリケーションのアイコンにドラッグ&ドロップした場合もやはり同様な動作になります。必要ならアプリケーションを起動し、そのアプリケーションに対して Open イベントを送り届けられます。Open イベントを受け取ったアプリケーションは、指定された文書ファイルを開くわけです。

文書ファイルのダブルクリックでは、その文書のクリエイターに関連したアプリケーションの起動しかできません。一方、ドラッグ&ドロップの場合、リソースをきちんと定義していれば、作成したアプリケーションではない関連付けられていないアプリケーションに対してもドラッグ&ドロップで開くことができます。

Open イベントを Java アプリケーションで受け付け

Open イベントは、Mac OS の API レベルで言えば、システムからのコールバックルーチンで実現しています。あらかじめ C/C++で定義された関数があって、その関数をシステムに登録します。つまり、Open イベントがあればこの関数を呼び出すようにという指定をまず行います。そして、アプリケーションに対して Open イベントが送られたなら、実質的には定義したコールバック関数がシステムから呼び出され、そこ

でファイルを開いたりするような処理を行います。もちろん、Java のアプリケーションでは同じやり方はできません。そこで、MRJ の環境下では、こうした AppleEvent でのやりとりがあったときに、定義したクラスのメソッドを呼び出すような仕組みをアプリケーションに提供しています。

考え方としては、Java のプログラムソースで定義するクラスに、Open イベントの処理機能を組み込むということを行います。そして、その機能を組み込んだクラスが、実際にイベントが発生したときに呼び出されるように、システムに対して登録を行うと考えます。そうすると、イベントが発生したときに既定のメソッドが呼び出されます。

まず、こうしたメカニズムが定義されたクラス群を利用できるようにするために、ソースの最初に次のような import 文が必要です。

```
import com.apple.mrj.*;
```

♦Apple Event を受け付けるクラスを定義

次に、ある特定のクラスが Open イベントを受け付けるように定義をします。その Open イベントを受け付ける機能は、MRJOpenDocumentHandler という interface として定義されたクラスに組み込まれています。このクラスをインプリメントすれば、そのクラスでは Open イベントを受け付けるようになります。

そして、実際に Open イベントが発生したときに呼び出されるのは、void handleOpenFile(File)というメソッドです。この名称のメソッドが、イベント発生時に呼び出されるというのが決まりです。したがって、以上のことから、まず、次のようなプログラムの大枠は決まってしまうということになるでしょう。

```
import com.apple.mrj.*;
import java.io.*;

public class Draggable implements MRJOpenDocumentHandler
{
    public void handleOpenFile(File targetFile)
    {
        /*    Open イベントに対応した処理    */
    }
}
```

ここで、クラス Draggable を定義していますが、MRJOpenDocumentHandler をインプリメントしているので、このクラスは Open イベントを受け付けることが可能になります。そして、Open イベントの発生時に実行されるメソッドの handleOpenFile も

定義しますが、public であり、戻り値はないので void を記述します。そして、引数は、java.io.File 型のデータが得られます。この引数は、もちろん、ダブルクリックしたり、アプリケーションにドラッグ&ドロップしてきたファイルを参照する File 型オブジェクトです。つまり、引数によってどのファイルを開くのかは分かるということです。

♦イベントを受け付ける

こうして定義したクラスは、Open イベントを受け付ける能力を身に付けているのですが、Mac OS システムは、このクラスにそういう能力があることをまだ知りません。そこで、Mac OS システムに対して、Open イベントの処理クラスはこれだということを知らせなければなりません。つまり、Open イベント処理クラスの登録を行うのですが、それには MRJApplicationUtil クラスある以下のようなクラスメソッドを利用する必要があります。

表 3-1　Open イベントを登録する MRJApplicationUtil クラスのメソッド

返値の型	メソッド	機能と利用方法
void	MRJApplicationUtil.registerOpenDocumentHandler(MRJOpenDocumentHandler handler)	引数に指定したクラスが Open イベントに応答するようにする【static】

この registerOpenDocumentHandler メソッドの引数は、定義としては、MRJOpenDocumentHandler 型のクラスへの参照ということですが、つまりは、MRJOpenDocumentHandler をインプリメントしたクラスが引数に指定されていればいいというわけです。そのクラスを Open イベントに対応するようにシステム側で手はずを整えるということになります。

実際に registerOpenDocumentHandler を利用するタイミングは、必ずしも以下に示すだけではありませんが、いちばん分かりやすいのは、クラスのコンストラクタで処理をしてしまうということです。つまり、クラス定義の大枠としては次のようになります。

```
import com.apple.mrj.*;
import java.io.*;

public class Draggable implements MRJOpenDocumentHandler
{
    public Draggable()
    {
        MRJApplicationUtil.registerOpenDocumentHandler(this);
    }
```

```
    public void handleOpenFile(File targetFile)
    {
        /*      Open イベントに対応した処理      */
    }
}
```

このように定義しておくと、クラス Draggable のインスタンスを生成すれば、そのインスタンスが Open イベントに対応できるようになります。もちろん、Open イベントが発生したときには、handleOpenFile が呼び出されます。

Open イベントに対応するということは以上の通りです。あとは、handleOpenFile の中を記述するのですが、すでに引数で File 型のファイルへの参照が得られています。これがもし C 言語で作成しているのであれば、AppleEvent の詳細な仕様を知った上で、イベントのパラーメータから実際のファイル情報を取り出すという部分をプログラミングしなければなりません。その点を比較すると、こうした Open イベント対応は Java で作ったプログラムの方がシンプルになっています。

3-3 その他の Finder 機能の受け付け

Open 以外の 3 つの Required Event について、その扱いを説明しましょう。また、MRJ 環境だけにある About イベントについても説明します。

Print イベントのハンドラ

Required Event のうち、Print イベントは、Finder 上で文書を選択し、「ファイル」メニューから「印刷」を選択したときに、アプリケーションに対して送り届けられます。このとき、文書に関連づけられたアプリケーションが起動し、そしてそのアプリケーションが文書ファイルを実際に印刷を行います。開いて印刷するか、直接印刷するかはアプリケーションに依存しますが、一時的にでも開くアプリケーションが一般的でしょう。

Open イベントほどは使われていないにしても、この Print イベントの対応についても、基本的には行わないといけません。MRJ 環境でも、Print イベントは Open イベントとほとんど同じように対処できるようになっています。

まず、Print イベントを受け付ける機能を持った MRJPrintDocumentHandler という intarface クラスが定義されています。Print イベントを受け付けるには、このクラスをインプリメントする必要があります。そうすると、Print イベントが発生したとき、handlePrintFile というメソッドが呼び出されます。

実際に Print イベントが発生したときにこのクラスのメソッドが呼び出されるようにするためには、以下のようなクラスメソッドを利用します。

表 3-2 Print イベントを登録する MRJApplicationUtil クラスのメソッド

返値の型	メソッド	機能と利用方法
void	MRJApplicationUtil.registerPrintDocumentHandler(MRJPrintDocumentHandler handler)	引数に指定したクラスが Print イベントに応答するようにする【static】

たとえば、あるクラスを、Print イベントだけに対応するとなると、次のような形式になるでしょう。

```
import com.apple.mrj.*;
import java.io.*;

public class Draggable implements MRJPrintDocumentHandler
{
    public Draggable()
    {
        MRJApplicationUtil.registerPrintDocumentHandler(this);
    }

    public void handlePrintFile(File targetFile)
    {
        /*    Print イベントに対応した処理    */
    }
}
```

つまり形式的には Open イベントも Print イベントもかわらないということになります。両方のイベントに対応するプログラムについては、2-4 節で具体例として紹介しましょう。

Quit イベントのハンドラ

Required Event のうち、Quit イベントは、終了のためのイベントです。もちろん、Command+Q によってユーザーが終了する時に内部的にこのイベント処理ルーチンを使っている可能性もありますが、このイベントの主目的は別のところにあります。それは、Mac OS を終了するときに、終了に先立ってアプリケーションを終了しますが、そのとき、Finder は起動している各アプリケーションに対して Quit イベントを送り届けます。また、インストーラを使ってソフトウエアをインストールする時に、インストーラ以外を終了させますが、そのときにも、Quit イベントは利用されているようです。

Quit イベントの処理についても、基本的に Open と同じように考えます。まず、Quit イベント処理機能は、MRJQuitHandler という intarface クラスによって定義されているので、このクラスをインプリメントします。そうすると、Quit イベントが送られてくれば、「public void handleQuit()」というメソッドを呼び出します。このメソッドの呼び出しには引数はありません。

そして、実際にイベントが発生したときにクラスのメソッドが呼び出されるようにするために、以下のようなクラスメソッドを利用して、クラスをシステムに登録します。

表 3-3　Quit イベントを登録する MRJApplicationUtil クラスのメソッド

返値の型	メソッド	機能と利用方法
void	MRJApplicationUtil.registerQuitHandler(MRJQuitHandler handler)	引数に指定したクラスが Quit イベントに応答するようにする【static】

実際の利用例は、3-4 節で説明しましょう。

New イベントは使われない

Required Event のうち、New イベントについては、MRJ 環境ではサポートされていません。New イベントについては、処理されなくても、Finder 操作への対応は十分にできるのです。

AppleEvent としての New イベントは、アプリケーションをダブルクリックして起動したときに必ず発生します。ただし、すでに起動している場合にはイベントは発生しません。アプリケーションを単に立ち上げた時、一般的なアプリケーションでは、名称未設定の新規の文書ファイルが開かれています。つまり、New ドキュメントを作るということで、これに対応した New イベントが定義されているというわけです。

もちろん、C や C++で作る時には New イベントによって新しい文書が開くように機能を作り込んでおきます。しかしながら、AppleEvent 処理をする意味はほとんどありません。起動時に新規ファイルを作るということであれば、イベントにたよらずに、プログラミングができるからです。スクリプトとして組んだ手順の中では、もちろん新しい文書を作るというニーズはあるでしょう。しかし、それは Finder の作業との整合性とは別の話になります。

以上のような理由で、もはや MRJ では New イベントの処理は不要だと考えた結果、取り入れられていないのでしょう。

About イベントのハンドラ

MRJ 環境独特のイベントとして、About イベントがあります。これは、アップルメニューのいちばん最初の項目である「○○について」「About ○○」という項目を選択したときに呼び出され、一般にはアプリケーションの名前や制作者、バージョンなどの基本情報を提供することに利用されています。こうした情報をダイアログボック

スで表示することもあり、一連の機能は「About ダイアログ」などとも呼ばれます。

C や C++では、この About ダイアログの表示はイベントとしては処理しません。メニューを選択した結果、アップルメニューの先頭項目であれば、About ダイアログを表示するというような機能の組み込みを行います。

なぜこのようなイベントが必要になるのかと言えば、それは Java VM という OS から離れたところで機能させるからだと言えるでしょう。Java のアプリケーションなどで、メニューを定義して利用できますが、そのとき、アプリケーションで使うメニューを定義します。MRJ ではそうしたアプリケーションでも、自動的にアップルメニューを最初に持ってきます。つまり、アップルメニューは自動的に用意されてしまうために、逆にプログラム内で選択したかどうかを取得する手段がなくなってしまうのです。そのために、MRJ が About ダイアログの表示要求があったことをアプリケーションに知らせる手段を用意しているものと思われます。

About イベントの処理についても、基本的に Open と同じように考えます。まず、About イベント処理機能は、MRJAboutHandler という intarface クラスによって定義されているので、このクラスをインプリメントします。そうすると、Quit イベントが送られてくれば、「public void handleAboutt()」というメソッドを呼び出します。このメソッドの呼び出しには引数はありません。

そして、実際にイベントが発生したときにクラスのメソッドが呼び出されるようにするために、以下のようなクラスメソッドを利用して、クラスをシステムに登録します。

表 3-4　Quit **イベントを登録する** MRJApplicationUtil **クラスのメソッド**

返値の型	メソッド	機能と利用方法
void	MRJApplicationUtil.registerAboutHandler(MRJAboutHandler handler)	引数に指定したクラスが About イベントに応答するようにする【static】

実際の利用例は、3-4 節で説明しましょう。

3-4　サンプルプログラムのソース

Required AppleEvent に対応したアプリケーションを作成してみました。テキストファイルを文書ファイルとして、Finder でテキストファイルをアプリケーションにドラッグ&ドロップすると、それを開くというものです。ソースファイルは 2 つです。

Draggable.java

ドラッグ&ドロップを受け付けるアプリケーションのクラスがこの Draggable.java です。ソースは以下の通りですが、これまでに紹介してきた MRJ がサポートする 4 つのイベントの処理を組み込んであります。

```
import com.apple.mrj.*;
import java.io.*;
import java.util.*;
import java.awt.*;

public class Draggable
    implements      MRJOpenDocumentHandler,
                    MRJPrintDocumentHandler,
                    MRJQuitHandler,
                    MRJAboutHandler     //各イベントの interface クラスをインプリメントする
{
    static public void main(String arg[])      //アプリケーションとして起動したときに呼び出される
    {
        System.out.println("Application Start");   //単にコンソールにメッセージを表示
        System.out.println(System.getProperty("test_prop"));
            //起動時にプロパティを指定した時の処理のチェック用（この章とは無関係）

        new Draggable();            //このクラスのインスタンスを作成して、
                                    //コンストラクタを呼び出す
    }

    public Draggable()      //コンストラクタ
    {
        MRJApplicationUtils.registerOpenDocumentHandler(this);
        MRJApplicationUtils.registerPrintDocumentHandler(this);
        MRJApplicationUtils.registerQuitHandler(this);
        MRJApplicationUtils.registerAboutHandler(this);
            //自分自身を、イベント処理クラスとしてシステムに登録する
    }

    public void handleOpenFile(File filename)      //Open イベントで呼び出される
```

```
    {
        System.out.println("Dragged "+filename.toString()); //コンソールにメッセージ
        TextViewer newViewer = new TextViewer(filename);
            //呼び出し時の引数のファイルを、TextViewer クラスを新たに用意して開く
            //TextViewer はウインドウにテキストを表示することが可能なクラス
    }

    public void handlePrintFile(File filename)      //Print イベントで呼び出される
    {
        System.out.println("Print Event "+filename.toString());     //コンソールにメッセージ
        TextViewer newViewer = new TextViewer(filename);      //TextViewer で開く

        /*      以下はプリントのための処理。TextViewer 側にメソッドとして定義する
            方がすっきりするかもしれないが、print や printAll をオーバーライドする
            こともあって以下のようにした。なお、印刷処理自体は不完全である。
        */
        PrintJob pj = newViewer.getToolkit().getPrintJob(newViewer,"A",null);
            //プリントジョブを用意する
        Graphics pg = pj.getGraphics();    //印刷のためのグラフィクスを取得
        if(pg != null)      {
            newViewer.printAll(pg);       //TextViewer を印刷
            pg.dispose();                 //解放するとプリンタに印字を行う
        }
        pj.end();    //ジョブの終了
    }

    public void handleQuit()      //Quit イベントで呼び出される
    {
        System.out.println("Quit Event Received");    //コンソールにメッセージ
        System.exit(0);           //アプリケーションを終了する
    }

    public void handleAbout()    //About イベントで呼び出される
    {
        System.out.println("Select About Menu");      //コンソールにメッセージ
    }
}
```

♦アプリケーションとして機能させる

Finder からの Required イベントを受け付け、かつアプリケーションとして機能させるような Java アプリケーションを作る場合の 1 つのパターンがこのソースにあります。まず、アプリケーションなので、main メソッドがどこかに必要です。イベントを受け付けるクラスに、アプリケーションで最初に実行される main を組み込むことにします。そこでは、自分自身をインスタンス化していますが、それによってコンストラクタが呼び出され、registerOpenDocumentHandler などのイベント処理クラスを登録するメソッドを使い、やはり自分自身をイベント処理クラスとして登録をしています。

別の方法としては、main メソッドを持つクラスと、イベント処理の interface クラ

スをインプリメントしたクラスを別々にしてもかまわないでしょう。作成するプログラムに応じて、臨機応変に対応すればいいことです。

♦各イベントでの処理

Draggable クラスをインスタンス化し、それをイベント処理クラスとして登録しているので、MRJ が管理する 4 つのイベントが発生したとき、Draggable クラスのメソッドが呼び出されることになります。

まず、About イベントが発生したときは、handleAbout メソッドが呼び出されます。ここでは簡単にするために、単にコンソールに文字列を表示していますが、通常はここでダイアログボックスを表示して、アプリケーションの説明などを表示するようにしたいところです。

Quit イベントが発生すると、handleQuit メソッドが呼び出されます。ここでも、記録を残す意味でコンソールに出力していますが、java.lang.System クラスで定義されているクラスメソッドである exit を利用して、アプリケーションそのものを終了させています。この exit の引数は、システムに返すエラーコードということになっていますが、Mac OS 環境ではそうした仕組みがないので、エラーなしを意味する 0 を指定しておくことしかできません。

Open イベントが発生すると、handleOpenFile メソッドが呼び出されます。引数には開くべきファイルが与えられています。もし、複数のファイルがアプリケーションにドラッグ&ドロップされれば、ファイルの数だけ handleOpenFile が呼び出されます。つまり、handleOpenFile では、複数のファイルを分離する処理は必要ありません。

HandleOpenFile が呼び出されると、コンソールにまずメッセージを残します。そして、ファイルの内容、つまりテキストをウインドウに表示するために TextViewer クラスをインスタンス化します。このクラスは後で説明します。

Print イベントも Open と基本的に同様ですが、プリントジョブに関する処理をここに記述しました。というのは、テキストをウインドウで表示する TextViewer は Frame を拡張しており、結果的には Component クラスのサブクラスになっています。したがって、すべてを印刷する printAll というメソッドが定義されているのですが、TextViewer ではそのメソッドでの印刷方法ではない手法を利用したいので、printAll をオーバーライドして書き直しています。ただ、印刷ジョブの取得からの処理は、TextViewer 側に移行した方がすっきりするのかもしれませんが、printAll の利用方法を本来の定義と一致させたために、ジョブの取得などを handlePrintFile 内に記述しました。

♦印刷時の処理

getPrintJob によって、Mac OS の印刷ジョブが開始され、「印刷」のダイアログボックスが表示されます。Mac OS では、印刷ジョブを取得し、そこでのグラフポートを取得して、グラフポートに描画することによって印刷結果を作成します。Java でも基本的には同様で、印刷ジョブの Graphics クラスのインスタンスを取得し、その Graphics オブジェクトに対して描画を行えば、印刷処理ができるという手はずです。描画は printAll メソッドで行い、dispose メソッドにより描画処理を終えたことをジョブに伝えることにより、実際にプリンタに対して印字処理が行われます。

TextVewer.java

このクラスは、テキストファイルの内容を読み込み、ウインドウを開いて、AWT のコンポーネントの 1 つである TextArea を配置し、そこに読み込んだテキストを表示するという機能を持っています。その意味で、テキストビューアです。TextArea をそのまま使っているので、テキストの折り返しはできません。コンポーネントをそのままつかっているため、編集作業もでてきてしまいます。保存の処理を組み込めば、テキストエディタになるかもしれませんが、今回のプログラムでは、Required イベント処理を念頭に置いているので最低限の処理しか組み込んでいません。

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;

public class TextViewer extends Frame       //ウインドウである Frame を拡張して定義
    implements WindowListener, ComponentListener
                                    //Window と Component のイベントを受け付ける
{
    File openFile;              //開いているファイルを記録
    TextArea viewText;      //読み込んだテキストを、TextArea コンポーネントで表示

    public TextViewer(File targetFile)  //コンストラクタ。開くファイルを引数に指定する
    {
        openFile = targetFile;  //開くファイルをメンバ変数に記録

        addWindowListener(this);    //Window イベントを受け付けるクラスとして登録
        addComponentListener(this);       //Componet イベントを受け付けるクラスとして登録

        show();                         //ウインドウを表示する
        setSize(300,200);               //ウインドウの初期サイズ
        setTitle(openFile.getName());       //ウインドウのタイトルは開くファイルのファイル名
        setLayout(null);                //レイアウト機能は無効にする
```

```
        viewText = new TextArea();  //TextArea を新しく用意し、参照をメンバ変数に記録
        viewText.setBounds(0,0,300,200);        //TextArea の位置と大きさを設定する
        add(viewText);                  //TextArea をウインドウに配置する

        //テキストをファイルから読み込む
        StringBuffer fText = new StringBuffer("");      //読み込んだデータを溜め込むバッファ
        String aLine;               //ファイルから読み込んだ 1 行分を保持する変数
        try    {
                LineNumberReader LNR = new LineNumberReader(new FileReader(targetFile));
                        //ファイルからの行読み込みを行う Reader を新たに用意する
                while(true) {       //ここは無限ループ
                        aLine = LNR.readLine();       //ファイルから 1 行読み込む
                        if(aLine == null)           break;
                                        //null だと最後まで読み込んだ。ループを抜け出す
                        fText.append(aLine + "¥n");  //読み込んだテキストをバッファに追加
                };
                LNR.close();       //ファイルをクローズする
        }
        catch(Exception e)      {
                System.out.println(e.getMessage());
        }
        viewText.setText(fText.toString());
                //ファイルから読み込んだテキストを TextArea に設定する
}

public void componentResized(ComponentEvent e)
                //ウインドウのサイズを変更した時に呼び出されるメソッド
{
        System.out.println("Window Resized");  //コンソールにメッセージ
        viewText.setSize(this.getSize());
                        //ウインドウの大きさに合わせて、TextArea の大きさも変更する
}

/*      以下のイベント対応メソッドは、特にここでは使用していない。
        Abstruct で定義されたメソッドなために、中身は空のままでも定義だけは必要になる
*/
public void componentMoved(ComponentEvent e)  {}      //ウインドウを移動した時
public void componentShown(ComponentEvent e)  {}      //ウインドウを表示した時
public void componentHidden(ComponentEvent e)  {}     //ウインドウを非表示にした時

public void windowOpened(WindowEvent e)   {}      //ウインドウを開いた時
public void windowClosing(WindowEvent e)   {}     //ウインドウを閉じている時
public void windowClosed(WindowEvent e)    {}     //ウインドウを閉じた後
public void windowIconified(WindowEvent e)  {}    //ウインドウを最小化したとき
public void windowDeiconified(WindowEvent e)      {}      //ウインドウを最小化から戻した時
public void windowActivated(WindowEvent e)        {}      //ウインドウがアクティブになる時
public void windowDeactivated(WindowEvent e)      {}      //ウインドウがデアクティブになる時

public void printAll(Graphics g)      //Component の printAll メソッドをオーバーライドして、
                                      //TextArea の中身を印刷するようにした
{
        String targetLine;
        int currentBase = 0;     //この変数でテキスト行のベースラインを指定する
        int LinePitch = 16;      //1 行の行ピッチ。この数値をベースラインに加える
```

```
            StringTokenizer eachLines = new StringTokenizer(viewText.getText(), "¥n", false);
                  //TextArea の中身を改行で区切る。つまり、1 行ごとをトークンとして取り出す
            while(eachLines.hasMoreTokens())      {      //トークンつまり行がまだあるなら
                  targetLine = eachLines.nextToken();     //行のテキストを取り出す
                  currentBase += LinePitch;               //行のベースラインを下に進める
                  g.drawString(targetLine, 0, currentBase);       //1 行分を印刷する
            }
      }
}
```

♦テキストビューアの設計意図

TextViewer は、ウインドウを使ってテキストを表示するので、まずベースになるのは AWT のウインドウを処理するクラスである Frame を拡張して定義しているというところです。そして、ウインドウの中に TextArea を確保して、そこにテキストデータを設定することでテキストを表示しています。

TextArea は、ウインドウと同じ大きさにしておきます。すると、ウインドウ全体は TextArea になってしまうので、あたかもエディタのような画面になるはずです。TextViewer のコンストラクタの最初の部分は、ウインドウの設定を行い、TextArea を用意しています。

そして、用意した TextArea に、テキストファイルの内容を読み込んで設定しています。特にチェックはしていないので、無理にテキスト以外のファイルを開くことも不可能ではありませんが、意味はないでしょう。

♦ウインドウイベントに対応する

ウインドウの大きさをユーザーが変更したときの処理も組み込んであります。まず、TextViewer クラスに、イベントリスナーの interface クラスをインプリメントしておき、TextViewer にイベントに対応する機能を組み込みます。そして、コンストラクタで、自分自身をイベント処理クラスであることをシステム側に登録します。このあたりは、JDK 1.1 のデリゲーションイベントモデルによって提供されている機能なので、JDK 1.1 以降でしか機能しないプログラムになります。なお、Required イベント処理も、デリゲーションイベントと同じように扱えるように、クラス設計がなされているものと思われます。

そして、ウインドウのサイズをユーザーが変更した時、クラス内の componentResized メソッドが呼び出されます。ウインドウのサイズについては、ユーザーのドラッグ作業を受け付けて自動的に変更されますが、そのままだと中にある TextArea のサイズは変更されません。そこで、componentResized の中では TextArea のサイズを、現在のウインドウの大きさと一致させることを行っています。ウインドウのサイズを変更す

る処理はここでは必要なく、自動的に行われています。

なお、やや余分ですが、コンポーネントとウインドウのイベントリスナーをインプリメントしています。そうすると、リスナーの interface クラスに定義されているすべての abstruct クラスをオーバーライドしなくてはならないので、使わないイベント処理のメソッドも、定義だけはしておかないとエラーになってしまいます。一方、MRJ の Required イベント処理は、Open や Print など、1 つのイベントに 1 つの interface クラスを定義しているため、イベントごとにクラスのインプリメントが必要になりますが、不要な処理メソッドはインプリメントしないでおけば、記述する必要がなくなっています。

♦印刷時の処理と制限

Frame クラスに対して printAll メソッドを実行することで、ウインドウ内に追加されたコンポーネントも含めて印刷が可能です。ただし、その場合だと、画面に見えるように印刷が行われるので、テキストがすべて見えるわけではありません。それではちょっとおもしろくないということもあって、少し違った形式のプログラムにしてみました。printAll をオーバーライドして、一般的なテキストエディタのように印刷することを考えました。

手順としては、TextArea から 1 行ずつ取り出して、それを Graphics に対して描画を行います。そのとき、テキストの描画位置を指定するため、現在の行のベースライン位置を変数で管理し、1 行の描画が行われるとそのベースライン位置を下にずらして次の行を印刷するということを行っています。詳しい処理内容ついてはコメントを参照してください。

しかしながら、現状では、まず、1 ページ分しか印刷されず、2 ページ以上に渡るような長いテキストでは、完全に印刷ができません。また、1 行の長さが横幅を超えるような長いものの場合、幅を超えたテキストは切れてしまいます。今回のプログラムは完全な印刷ルーチンを作成することが目的ではないため、簡易なプログラムで済ませてあります。

なお、印刷処理については、MRJ 2.0 では意図通りに機能しますが、MJR 2.1ea2 では行の分割が StringTokenizer でうまくなされず、1 行分しか印刷されない結果となってしまいます。

作成したリソース

Finder 上でテキストファイルのドラッグ&ドロップを受け付けるためには、テキス

トファイルを文書として認識するように BNDL リソースが必要になります。そのためのリソースを、作成したアプリケーションに組み込まなければなりません。組み込む方法については、第 4 章で説明しますが、ここで紹介したサンプルを機能させるためには、以下のようなリソースを作成しています。ここでは ResEdit を使って作成をしました。

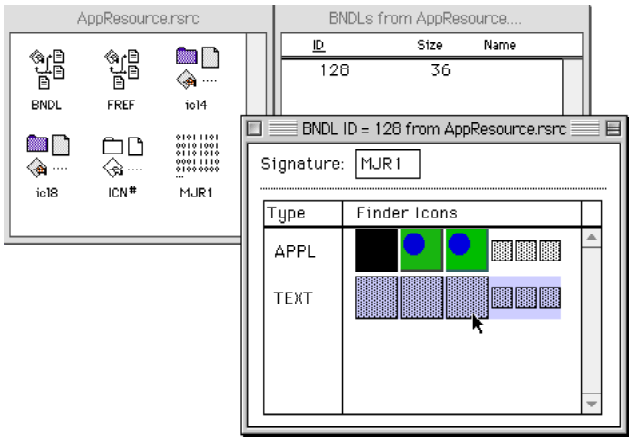

図 3-1　作成したリソース

リソースファイルを新たに用意し、そこに BNDL リソースを作成します。そして、Signature にクリエイター（ここでは MJR1）を指定します。そして、ファイルタイプとしてアプリケーションの APPL と、テキストファイルの TEXT を追加します。アプリケーションの方ではアイコンが表示される部分をダブルクリックして、アイコンを適当に作っておきます。アイコン関連の 4 つのリソースや、「MJR1」リソースは結果的に自動的に作られるので、作業としては BNDL リソースだけを作ればいいということになります。

実行結果

こうして作成したアプリケーションを実際に機能させてみます。テキストファイルをアプリケーションにドラッグ&ドロップします。また、テキストファイルのクリエイターを MJR1 に変更すればダブルクリックして開くこともできます。

図 3-2　作成したアプリケーションの「Draggable」にテキストファイルをドラッグ&ドロップした

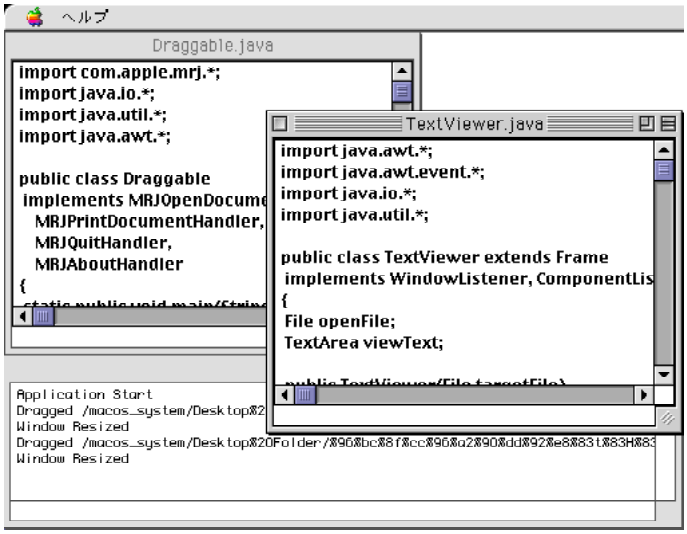

図 3-3　Draggable が起動して、それぞれのテキストファイルをウインドウで表示した

LOCUS Report

第4章　アプリケーションを作成する

● Mac OS で実際に利用可能な各種ツールを利用して、アプリケーションを作成する方法を説明します。

● アプリケーションとして実行可能なソースの作成方法は、第 3 章で説明しました。ここでは、その規則に従ったソースがすでに存在する、あるいは作成するということを前提にしています。

● MRJ SDK を利用した方法、そして市販のツールとして Visual Cafe Ver.2.0 と Code Warrior Pro4 を使った方法を説明します。

4-1　MRJ SDK を利用したアプリケーション作成

MRJ SDK は事実上フリーで入手できるツールなので、気軽に利用できるということもありますが、Mac OS での開発環境の一種のリファレンスにもなっています。アプリケーション化の機能では、MRJ SDK でないとできないこともあり、基本的には別の開発ツールを使っていても、MRJ SDK を利用する機会もあり得るでしょう。98 年 11 月現在、MRJ 2.1 EA3 と、MRJ SDK 2.1EA2 が実行環境と開発環境の最新版です。これらを利用した開発作業を紹介しましょう。

MRJ SDK のセットアップ

MRJ SDK についての概略は、第 1 章で解説をしました。実行環境の MRJ とは別に配付されている開発ツール群が MRJ SDK です。Apple 社の MRJ のページから、つねに最新版をダウンロードできるようになっています。MRJ SDK をセットアップすると、起動ディスクのルートに「MRJ SDK 2.1」フォルダが作成され、中身は次のようになっています。アプリケーション作成では、JBindery と Tools フォルダの内容を使います。

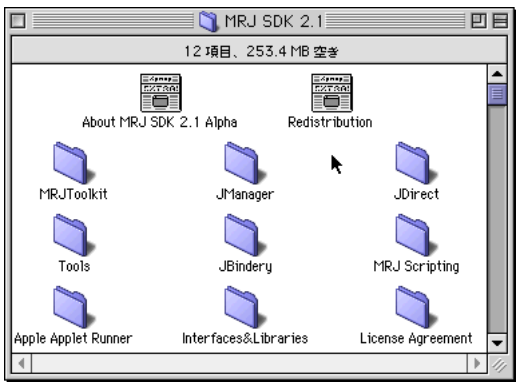

図 4-1　インストールした MRJ SDK のフォルダ

MRJ SDK のインストーラは、単にツール類をハードディスクに展開するだけではありません。これらのツールを利用するのに必要なライブラリやクラスファイルについてもシステムフォルダにインストールします。システムフォルダ内の「機能拡張」フォルダにある MRJLibraries フォルダには、まず、MRJSDKLib という名前のライブラリが組み込まれます。さらに、その中にある MRJClasses フォルダには MRJSDKClasses.zip と Properties.zip というファイルが組み込まれます。

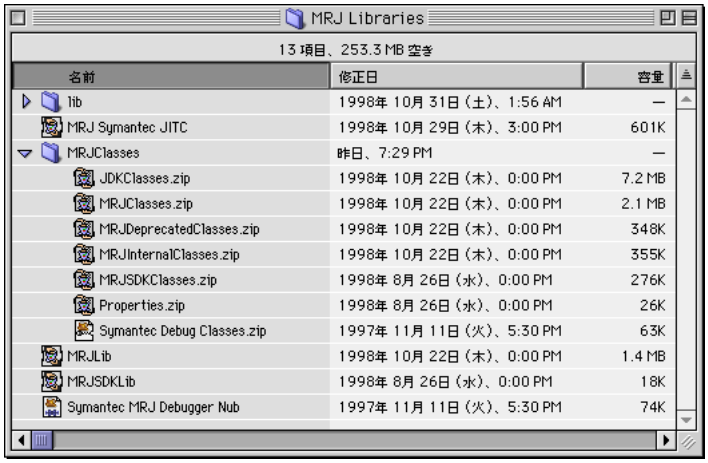

図 4-2　MRJLibraries **フォルダにもファイルはインストールされる**

これらのファイルについての詳細はどこにも解説されていませんが、MRJSDKClasses.zip は、コンパイラである javac によって利用されるクラスファイルなどが含まれており、開発ツールを実行するのに必要になります。このファイルがないと、コンパイラは起動時にエラーメッセージを表示してコンパイルは行われません。Properties.zip については、javac 関連のプロパティと思われる設定項目が並んでいます。

なお、これら MRJ SDK によってインストールされるファイルは、MRJ 自体ではインストールされません。実行環境をいろいろ切り替えたりしたときに、この開発ツールに必要なファイルがないとか、大きくバージョンが違うものがインストールされてしまうということも考えられます。基本的には、MRJ と MRJ SDK のバージョンは一致させておく必要があります。MRJ を切り替えて使っている人は開発ツールで必要なファイルの扱いも注意する必要があります。

コンパイラの起動

Java コンパイラは、MRJ SDK 2.1:Tools:JDK Tools フォルダにある javac というアプリケーションです。すでに、テキストエディタなどでプログラムが作成されていると

して、以下の説明を行います。プログラムソースのファイルは、テキスト形式で、ファイル名はその中で定義しているクラス名に.java を付けたものである必要があります。これは Java 言語での決まりとなっています。

コンパイルは、ソースファイルを javac にドラッグ&ドロップするだけです。手順としては次のようになります。まず、javac とソースファイルの両方が Finder 上で見えるようにウインドウを開けておきます。ここでは、Draggable.java と TextVewer.java の 2 つのソースファイルをまとめてコンパイルすることにします。これらのファイルを選択して、javac にドラッグ&ドロップします。

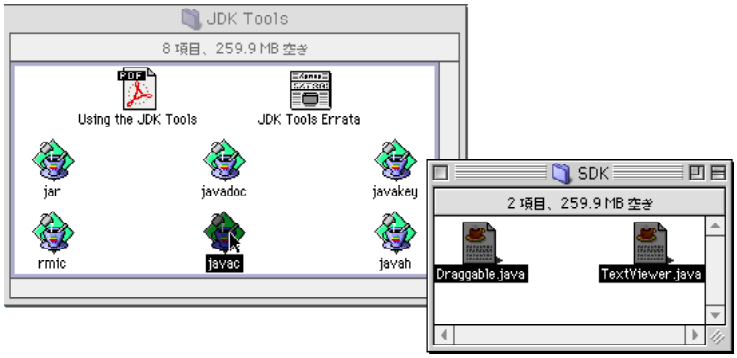

図 4-3　ソースファイルを javac にドラッグ&ドロップする

すると、javac が起動して、次のようなダイアログボックスが表示されます。Source Files に、ドラッグ&ドロップしてきたソースファイルのフルパスが表示されているのを確認して、Do Javac ボタンをクリックすれば、コンパイルが行われます。他のオプションについては後でまとめて説明しますが、通常はそのままの設定でかまわないでしょう。

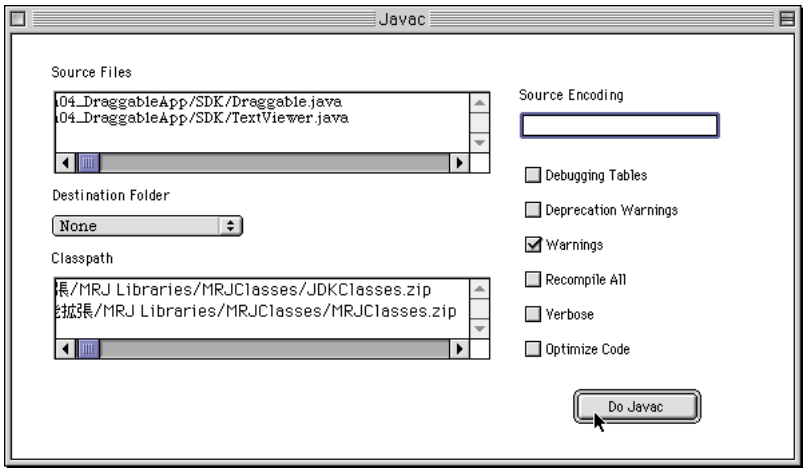

図 4-4　　コンパイラの設定ダイアログボックスが表示される

その後、コンパイルが行われます。コンパイル状況は javac でのコンソールに出力されますが、Done と表示されればコンパイルは終了です。既定の状態では、コンパイルするファイルのパスや、致命的なエラーが表示されるようになっています。この場合は、ソースファイルと同じフォルダに、コンパイルした結果である Draggable.class と TextViewer.class の 2 つのファイルが新たに作成されています。

図 4-5　　コンパイルした結果、クラスファイルが作成された

javac は終了するまで起動したままになります。同じファイルを連続してコンパイルするときには、単に Do Javac ボタンをクリックするだけでかまいません。

なお、javac は、日本語の文字列を Shift-JIS コードで含むようなソースファイルも正しくコンパイルして、実行したときにきちんとソースで指定した文字列が表示されます。各国のシステムにきちんと対応しているようです。

ただし、文字列中にある¥の記号は、バックスラッシュとしては処理されません。

これは非常に不便です。¥をバックスラッシュとして解釈しないと、文字列中でのエスケープ文字の指定ができなくなってしまいます。おそらく、UNICODE では¥とバックスラッシュが別のコードに割り当てられていて、まじめに文字コードを解釈し、きちんと文字コード変換をしているための副作用でしょう。¥n は「¥n」という文字列そのものになります。¥"は本来はダブルクォーテーションを文字列中に指定するために利用できるはずなのですが、¥で文字列が終わりであると解釈してしまいます。これを避ける方法や設定については現状では不明です。たとえば、byte 配列にダブルクォーテーションの文字コードを入れておき、それをもとに String オブジェクトを生成してつなげてやるようなことが必要になるかと思われます。

javac コンパイラのオプション

javac の設定ダイアログボックスではさまざまなオプション設定ができますが、一般的なコンパイルでは特に変更する必要はないことが多いでしょう。ここでは、これらのオプションの機能を説明しておきます。

まず、Source Files は、コンパイル対象のソースファイルです。これらは一般には javac へドラッグ&ドロップしたものが自動的に設定されています。Source Files の項目に、Finder で javac にドラッグ&ドロップしたソースファイルのフルパスが表示されているのを確認して、Do Javac ボタンをクリックすれば、それだけでコンパイルされます。コンパイル対象のソースファイルを増やした場合には、Source Files のテキストエリアでソースのパスをキータイプして追加することでも可能です。しかしながら、Finder 上で、ソースファイルを javac にドラッグ&ドロップすることでも、そのファイルを Source Files に追加することができるので、こちらの方が手軽で確実です。

なお、javac は、日本語の文字列が Shift-JIS コードを含むようなソースファイルも正しくコンパイルして、実行したときにきちんとソースで指定した文字列が表示されます。各国のシステムにきちんと対応しているようです。

Distination Folder は、コンパイルした結果のクラスファイルを保存するフォルダを指定します。JDK の javac の-d オプションに相当します。None が選択されていれば、ソースファイルと同じフォルダに作成します。ポップアップメニューから Select Directory を選択すると、ダイアログボックスで保存先のフォルダを選択することができます。複数のフォルダを設定して、選択することができますが、このフォルダは javac を終了するとその時点で忘れ去られます。なお、JDK の javac では、package 文の指定があると、それに応じた相対パスにコンパイル結果のクラスファイルが作成されます

が、MRJ SDK の javac では package があっても特にそれをフォルダに展開することは行いません。

Classpath は、参照するライブラリのパスを指定します。JDK の javac の-classpath オプションに相当します。たとえば AWT などのライブラリを使っていると、そのクラスのメソッドなどの参照を、ここで指定したクラスパスから行います。既定値として、MRJ での実行に必要なクラスファイルへのパスが設定されています。必要なら、このテキストエリアにキータイプして追加指定します。あるいは、ファイル名が.zip で終わるアーカイブ形式のライブラリファイルを、Finder 上で javac のアイコンにドラッグ&ドロップすることで、Classpath にそのライブラリファイルを追加することもできます。

Source Encode ではソースファイルのエンコード方法を指定します。JDK の javac の-encoding オプションに相当します。指定するとすれば、EUCJIS あるいは SJIS のようなテキストを指定しますが、指定がない場合には、デフォルトのエンコーディングになるようです。実際、通常とおり Mac OS 上で作った Shift-JIS の漢字を含むソースは正しくコンパイルされるようなので、普通に Mac OS 上のテキストエディタを使って作ったルのエンコード方法を指定します。指定するとしたら、EUCJIS あるいは

Debugging Tables のチェックボックスでは、デバッグのために、ソースコードの行番号情報を残すという設定です。MRJ SDK 自体にはソースコードデバッガがないので、これを利用することは基本的にはないと思います。JDK の javac の-g オプションに相当します。

Deprecation Warnings のチェックボックスを指定すると、たとえば Component クラスの reshape メソッドのように deprecate された（別のメソッドに置き換えられた）ものをソース中で利用している場合に警告を出します。JDK の javac の-deprecation オプションに相当します。

Warning は警告のメッセージを表示するかどうかを指定しますが、チェックが入っていれば警告を出力します。JDK の javac の-noworn オプションに相当しますが、チェックを付けていないことと、-noworn オプションを指定することが対応しています。

Recompile All は、ソースファイルが依存関係にあるソースもまとめてコンパイルするというオプションです。JDK の javac の-depend オプションに相当します。

Verbose は、コンパイル中のメッセージをコンソールに出力するオプションです。ロードしているクラスの情報などが逐一メッセージされます。JDK の javac の-verbose オプションに相当します。

Optimize Code は、最適化コンパイルを行うオプションです。JDK の javac の-O オプションに相当します。final や static、private メソッドをインライン展開することによる最適化を行います。したがって、実行速度は向上することが期待できますが、ファイルサイズは大きくなります。ただし、ソースが得られる部分だけをインライン展開します。デバッグのオプションは自動的にオフになります。

なお、MRJ SDK の javac では、JDK の javac にある-J オプション（Java インタープリターに渡す引数）は利用できません。JDK の java ツールは MRJ SDK には存在しないので、コンパイラで利用することはないとしているようです。

アプリケーションファイルの作成

javac を利用してクラスファイル、すなわち、Java のソースをコンパイルしたバイトコードのファイルができあがりました。そのファイルから、Finder 上で実行できるようなファイルを生成するのが JBindery です。JBindery は、クラスファイルの実行ということに加えて、実行可能なアプリケーションを作るという機能があり、ここでは後者の方法を説明しましょう。

Mac OS 向けのアプリケーションには、最低限、Finder 上でのアイコンの表示と、ドラッグ&ドロップによってファイルを開くための設定が必要です。こうした設定は Mac OS 上ではリソースとして管理されていることですが、Jbindery ではすべてを直接的にはできません。そこで、ResEdit を用いてリソースを作成し、そのリソースを JBindery で作成するアプリケーションに含めることで、Mac OS 上での必要な設定を加ます。

ここで作成するアプリケーションのクリエーターは、MJR1 ということにしておきます。そして、ここで追加する設定は、テキストファイルのドラッグ&ドロップを可能にするということです。

♦リソースの追加

リソースの作成は、ResEdit で行うのが一番手軽でしょう。もちろん、テキストでソースを記述して、MPW の Rez を使ってリソースファイルを作ってもかまいませんし、Resourcerer のようなリソース作成ユーティリティを使ってもかまいません。

ここでは、AppResource.rsrc という名前のリソースファイルを ResEdit で作成しました。作成した内容は以下の通りです。

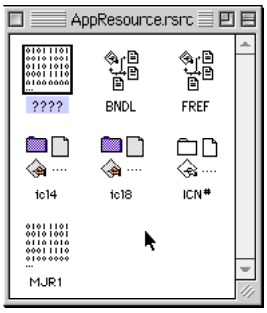

図 4-6　作成したリソースファイル

リソースファイルの作成については概略だけを説明しておきます。図では 7 つのリソースの種類がありますが、手順の上では、BNDL リソースを作成するだけで、他のリソースは作成されることになります。BNDL リソースは次の図のようなものを作成しました。アイコンは、適当にデザインしたものです。

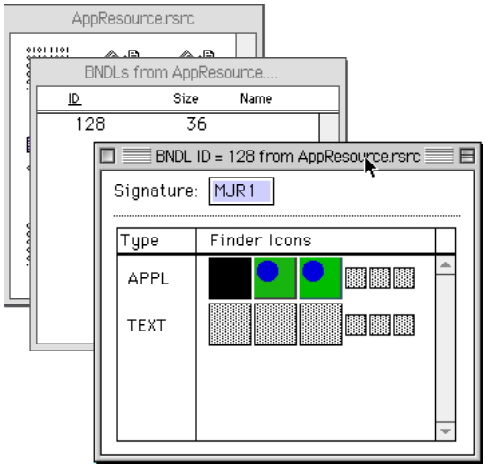

図 4-7　作成した BNDL リソース

リソースファイルの作成手順をまとめておきます。

1. **ResEdit を起動します。**
2. **最初に開くファイルを指定するダイアログボックスが出てきますが、そこで New ボタンをクリックして新しいリソースファイルを作ります。New ボタンをクリック後、さらに新しいファイルのファイル名と保存フォルダを指定するダイアログボックスになるので、Java のプログラムソースと同じフォルダに、AddResource.rsrc という名前でファイルを作成しておきます。**
3. **Resource メニューから Create New Resource（Command+K）を選択します。**
4. **ダイアログボックスが表示されるので、BNDL を選択して OK をクリックしま**

す。

5. 新しく BNDL リソースが定義されました。まず、Signature に、アプリケーションのクリエーターである MRJ1 をキータイプします。
6. Resource メニューから Create New Type（Command+K）を選択して、ファイルタイプを新たに付け加えます。
7. 付け加えられた最初のファイルタイプは、APPL にします。これにより、アプリケーションのアイコンが定義できます。
8. 再度 Resource メニューから Create New Type（Command+K）を選択して、ファイルタイプを新たに付け加えます。2 つ目のファイルタイプは TEXT にします。これによって、テキストファイルのドラッグ&ドロップを受け付けます。また、文書ファイルを適すとファイルとして作った時のアイコンも定義できます。
9. APPL の右側にある Finder Icon はまだ定義されていません。ここをダブルクリックします。そしてダイアログボックスで New ボタンをクリックすると、新しくアイコンを定義するダイアログボックスが表示されます。ここでアプリケーションのアイコンを定義します。
10. 同様に TEXT に対するアイコンも定義します。
11. 保存して、ResEdit を終了します。

♦JBindery の起動

ソースファイルからコンパイルしたクラスファイル、そしてリソースファイルが用意できたら、JBindery を起動します。JBindery はアプリケーションの起動のために利用するので、ここでは、main メソッドがある Draggable.class ファイルを JBindery のアプリケーションにドラッグ&ドロップします。JBindery は、MRJ SDK 2.1 フォルダの JBindery フォルダにあります。

図 4-8　実行するクラスファイルを JBindery にドラッグ&ドロップする

こうして起動すると、JBindery の設定ダイアログボックスが表示されます。まず、じっこうするクラスは、Command のアイコンのページにある Class name のテキストボックスに自動的に設定されているはずです。書き換えてもかまいませんが、ドラッグしてきたクラス名が自動的に設定されるので、通常はそのまま利用することになるでしょう。

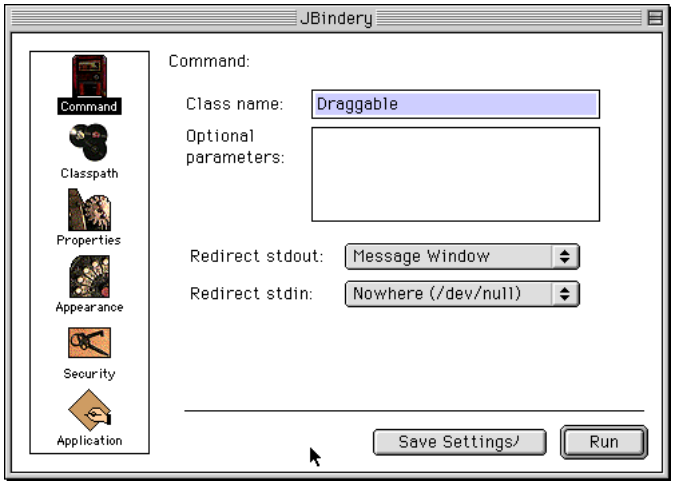

図 4-9　JBindery の Command の設定ページ

JBindery の各種設定については後で説明をします。ここでは、まず、目的となるアプリケーション作成に必要な設定だけをまとめておきます。

♦クリエータの設定

次に、設定ダイアログボックスの左側で、Application のアイコンを選択してクリックします。ここでは次の図のようなダイアログボックスが表示されます。まず、Creator

のところには、設定するクリエーターである MJR1 をキータイプします。そして、Marge resources from のチェックボックスをオンにすると、リソースファイルを選択するダイアログボックスが表示されます。そこで、作った AddResource.rsrc を指定して、含めるリソースとして設定します。

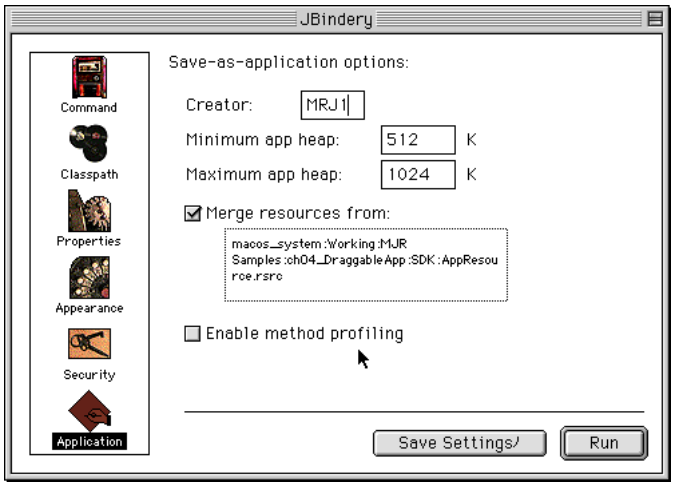

図 4-10　JBindery の Application の設定ページ

そして、Save Setting ボタンをクリックします。すると、保存するファイル名を指定するダイアログボックスが表示されます。ボタン名は「設定を保存」のような意味ですが、これによって作成されるのアプリケーションです。ダイアログボックスでは「Save as Application」のチェックボックスが設定されているのを確認してください。つまりここでは、JBindery での各種設定を記録したアプリケーションファイルを作ると解釈すればいいのかもしれません。

図 4-11　アプリケーション TV を作成した

ここで、作成したアプリケーションファイルには、クラスファイルの内容は含まれ

ていません。いろいろと試してみたところでは、アプリケーションには、実行するクラスファイルのファイル名と、それが存在するフォルダのエイリアスを記録しているようです。

まず、アプリケーションファイルだけを別のフォルダに移動しても、実行はできます。このアプリケーション自体はどこにあってもかまいません。もちろん、MRJ が組み込まれたシステムでなければなりません。

一方、この場合の、Draggable.class と TextViewer.class の 2 つのクラスファイルは、アプリケーションを作成したときに設定したフォルダから別のフォルダに移動すると、アプリケーションは実行できなくなります。ただし、そのフォルダを別のフォルダに移動しても実行はできるので、クラスファイルのあるフォルダのエイリアスを、JBindery が作成するアプリケーションで記録管理しているものと結論付けることができます。

ただ、そうなると、作成したアプリケーションを配付する場合の問題があります。それについては次に議論します。

クラスパスの設定

JBindery は、「実行するクラスファイルがどこにあるか？」という情報をファイルの中で管理していることになります。実行時に管理すべきクラスファイルは大きく分けて 2 通りあります。1 つは、AWT などの Java の標準ライブラリのクラスファイルです。こうした標準のライブラリについては、実行環境の MRJ で管理されているので、何も指定しなくても、正しく参照するようになっています。これら、基本的なライブラリは、MRJLibraries フォルダにあるのですが、このフォルダ内のクラスファイル JDKClasses.zip と MRJClasses.zip の検索は何も設定しなくても行われるということです。

一方、実行するアプリケーションやあるいはそこから利用する独自のクラスなどがどこにあるかを記述する必要があり、JBindery の Classpath の設定でそれができるようになっています。設定ダイアログボックスの左側にある Classpath のアイコンをクリックして設定できるようになっています。

次の図は、Draggable.class をドラッグ&ドロップして JBindery を起動した時の状況ですが、既定値では、ドラッグ&ドロップしてきたクラスファイルが存在するフォルダが、Classpath として自動的に設定されています。もし、追加で設定したいのであれば、Add Folder ボタンをクリックして、ダイアログボックスでフォルダを指定して、追加を行います。

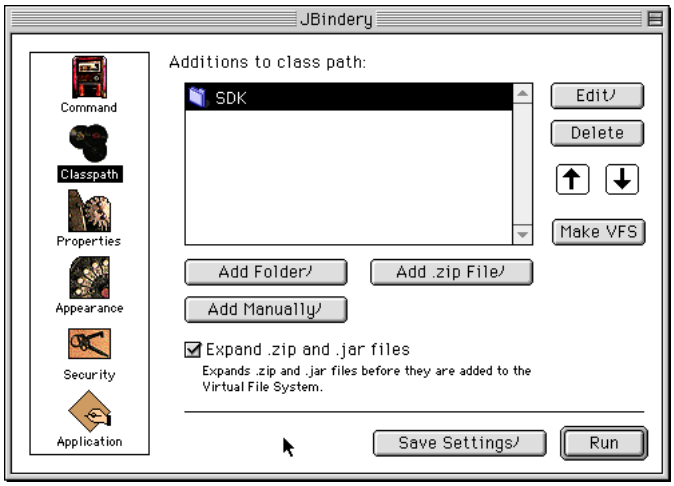

図 4-12　JBindery **の** Classpath **の設定**

また、Add .zip File をクリックすると、zip 形式にパッケージされたクラスファイルを追加することができます。これについても、ファイル自体のエイリアスを Classpath として追加します。

なお、Classpath の追加は、Path のリスト部分に、Finder からフォルダやファイルをドラッグ&ドロップしても行えます。フォルダや zip ファイルはそのまま追加されますが、.class ファイルをドラッグ&ドロップすると、そのファイルが含まれるフォルダが Path に追加されます。

♦アプリケーションとの相対ディレクトリを指定

クラスが存在するフォルダのエイリアスが記録されるとなると、他のマシンへファイルを移動する場合などでは、そのままでは問題になるでしょう。そのために、クラスファイルのありかを、JBindery で作成するアプリケーションとの相対パスで指定するという方法を取ることができます。

JBindery の Classpath の設定で、Add Manually ボタンをクリックします。次のようなダイアログボックスが表示されます。ここではパスをキータイプで入力できることを意味します。

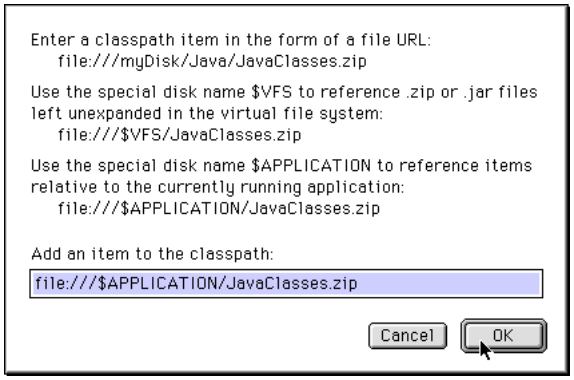

図 4-13　パスをキータイプして指定できる

ここで、$APPLICATION という仮想的なパスを記述することができます。たとえば、「file:///$APPLICATION/MyClass.java」で、アプリケーションと同じフォルダにある、MyClass.java というクラスファイルを参照します。「file:///$APPLICATION/Classes」なら、たとえば、アプリケーションと同じフォルダにある Classes フォルダということになり、その場合は Classes フォルダにあるクラスファイルも Classpath に加わって参照が可能になるということです。

アプリケーションで利用するファイルを相対パスで指定する方法だと、開発時にも同様な形態でファイルを配置しておくことができます。たとえば、new File("/$APPLICATION/") によって、アプリケーションの存在するフォルダを取得できるので、そこを手がかりにファイルを参照するプログラムを作成できます。また、デバッグ時と実行時で同じようなフォルダ構成にできるでしょう。

単独で実行できるファイルの作成

JBindery を使って単独のファイルで実行可能なアプリケーションも作成できます。そのためには、VFS（Virtual File System）という機能を使います。これは、ある 1 つのファイルが仮想的な 1 つのボリュームのように扱われるという機能です。その 1 つのファイル内にクラスファイルやあるいは利用している画像ファイルなども含めてまとめてしまうことができます。このファイルの中にあるファイルやフォルダは、「file:///$VFS/」という URL で参照することができます。

VFS を利用したアプリケーションを作成するには、たとえば次のように作業をします。

1. **必要なファイルがすべて入ったフォルダを用意します。フォルダ内に、クラスファイルやあるいは利用する画像ファイルなどを入れておきます。フォルダ内**

にフォルダがあってもかまいません。

2. **main メソッドが含まれるクラスを JBindery にドラッグ&ドロップして、JBindery を起動します。**
3. **クリエーターの設定や、追加するリソースなど、必要な設定を行います。**
4. **Classpath のページを開きます。そこには、手順 1 で用意したフォルダが Classpath に設定されいてるはずです。**
5. **そのフォルダを一覧で選択して、Make VFS ボタンをクリックします。すると、そのフォルダが VFS のルートとなります。**
6. **Save Setting ボタンをクリックして、アプリケーションを作成します。**

プログラム作成時、$VFS によって、基本的にはアプリケーションの存在するフォルダが参照できるようではあります。たとえば、new File("/$VFS/") で、File オブジェクトが生成されます。しかしながら、MRJ では、$VFS による参照がなぜかできなくなっています。問題点として報告されているようですが、MRJ 2.1 EA3 では直っていないようです。プログラム中で別のファイルを利用するような場合だと、デバッグと実行アプリケーションの両立にやや工夫が必要になるかもしれません。

システムプロパティとパラメータの利用

Jbindery では、起動時にアプリケーションに対して情報を与える機能を提供していますが、これはもともとは、JDK の Java インタプリタ、つまり Java アプリケーションの実行ユーティリティである java でのコマンドラインの機能に対応したものです。java はコマンドラインの引数を取るタイプのアプリケーションですが、Mac OS では OS 自体にコマンドラインという考え方はありません。そこで、JBindery では、コマンドラインが必要な Java アプリケーション向けに、コマンドラインを指定することができるようになっています。JBindery の Command のアイコンを選択したページで、Optional Parameters というテキストボックスがあります。そこにスペースで区切ったコマンドラインを指定することができます。指定したコマンドラインの各パラメータは、main メソッドの引数である arg に配列としておさめられます。

また、JDK の java コマンドでは、起動時に-D オプションをつけることで、システムプロパティを追加することができます。JBindery でもその機能は使えるようになっています。JBindery の Properties のアイコンのページで、起動時に追加するシステムプロパティの設定ができます。一覧の下の部分の「=」という記号がありますが、その左側にプロパティ名、右側にプロパティの値を入力して、Add ボタンをクリックすると、プロパティとして設定されます。ここで指定したプロパティは、システムプロ

パティを取得するクラスメソッドの System.getProperty("プロパティ名")で取得できます。

図 4-14　JBindery **の** Properties **の設定**

この章のサンプルプログラムでは、コマンドラインのパラメータと、システムプロパティが実際に正しくアプリケーションに渡されたかを、その値をコンソールに書き出すことで確認しています。以下のコンソールは、Optional Parameters に「aaaa bbbb cccc dddd」と指定し、前の図のようにシステムプロパティを設定して実行した結果です。

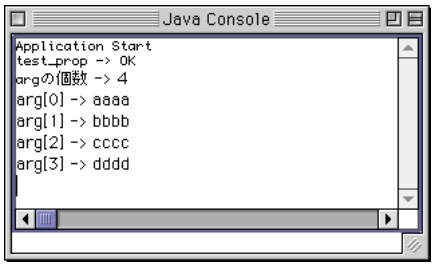

図 4-15　**プロパティとパラメータの取得結果**

その他の JBindery の設定

JBindery の Appearance のページでは、次の図のように、ウインドウの背景色や、サイズボックスを表示するかどうかを指定できます。背景色を変更するには、Window Background color の右にあるボックスをクリックします。すると、カラーピッカーのウインドウが表示されるので、そこで設定できます。

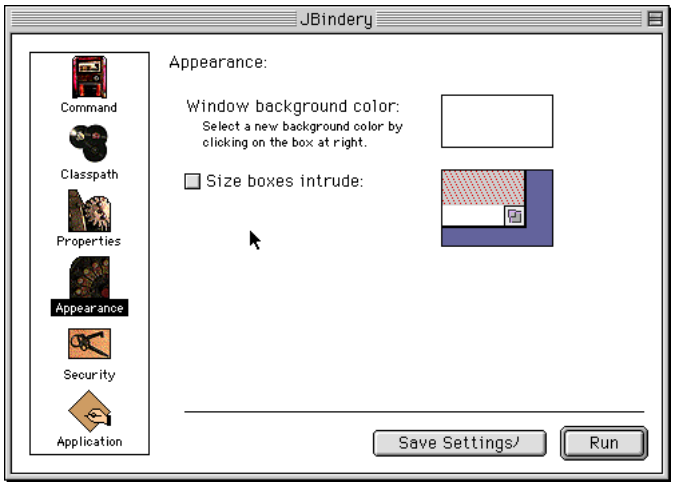

図 4-16　JBindery **の** Appearance **の設定**

JBindery の Security のページでは、バイトコードのチェックやあるいはプロクシサーバーの設定、ファイアウォールの設定などがあります。ただし、ネットワーク経由で得られたバイトコードのチェックに関しては、Verify bytecodes のチェックに関わらずに実行されるようになっています。

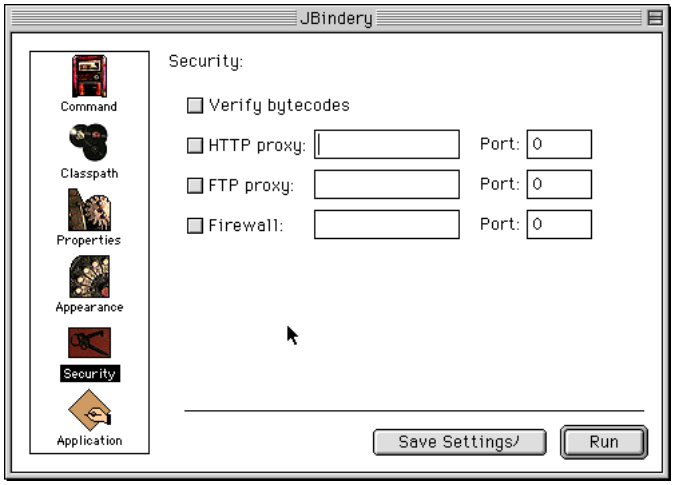

図 4-17　JBindery **の** Security **の設定**

4-2　Visual Cafe を使ったアプリケーション作成

Visual Cafe Ver.2.0 を使ったアプリケーション作成方法を説明しましょう。Visual Cafe の最新版は Macintosh 向けには発売されていませんが、Ver.2.0 でも MRJ 2.1 でのプログラム作成は可能です。ただし、EA3 段階ではデバッグ機能が利用できなくなっています。

プロジェクトの用意

Visual Cafe では、アプリケーションやあるいは Finder でのドラッグ&ドロップが可能なアプリケーションを作成するもとになるプロジェクトが利用できますが、ここでは設定を確認する意味も含めて、空のプロジェクトからアプリケーション作成をしてみます。

Visual Cafe を起動すると、最初はアプレットを作るプロジェクトが自動的に用意されていますが、そのプロジェクトファイルを閉じておきます。そのときに保存するかどうかをたずねてきますが、単に閉じるだけであれば、保存する必要はないでしょう。

まず、プロジェクトを用意しますが、「ファイル」メニューの「新規プロジェクト」（Command+shift+N）を選択します。すると、次のように、プロジェクトのテンプレートを選択するダイアログボックスが出てくるので、ここでは Empty Project を選択しておきます。

図 4-18　プロジェクトを選択する

すると、プロジェクトのウインドウが開きます。プロジェクトは適当な段階で、「ファイル」メニューの「保存」（Command+S）などを利用して、ファイルに保存しておきます。通常は、プログラムソースと同じフォルダあたりにプロジェクトのファイルを保存しておくことになるでしょう。もちろん、ソースファイルが多くなるなら、ソ

ースファイルだけをどこかのフォルダにまとめておくことで、整理がつくようになるかもしれません。

ソースファイルの作成と登録

Visual Cafe は統合開発ツールなので、テキストエディタも機能に含まれています。Java のキーワードに色をつけるなどして、プログラムを見やすく表示することができます。

新しくソースプログラムを作成するのであれば、「ファイル」メニューの「新規」（Command+N）を選択して、テキスト編集のウインドウを表示します。そして、そのウインドウに、プログラムを入力します。通常はそこで定義したクラス名に.java を付けたファイル名で保存しておきます。そうすると、「プロジェクト」メニューの「○○の追加」（○○はファイル名）という項目が選択できるようになります。このメニューを選択すると、プロジェクトに編集中のソースファイルを登録することができます。

あるいはすでにソースファイルが作成されているのであれば、「挿入」メニューから「ファイル」を選択します。そしてダイアログボックスで追加するファイルを指定して、プロジェクトにソースファイルを追加します。

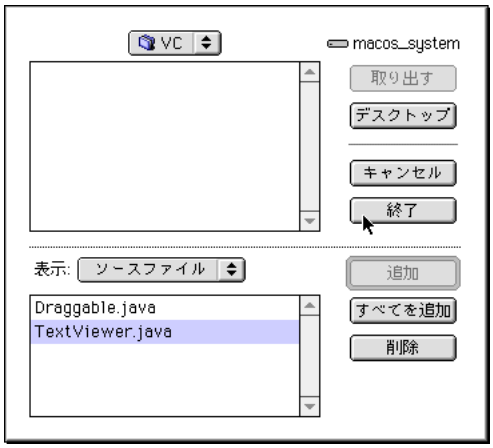

図 4-19　追加するファイルを指定するダイアログボックス

ダイアログボックスの上半分では、ボリューム内にあるファイルを表示しますが、ここでは、ファイル名が.java や.jar などで終わるなど、プロジェクトに追加可能な形式のファイルしか一覧されません。追加したいファイルが存在するフォルダを表示して、そのファイルの項目を選択し、「追加」ボタンをクリックすると、下側のリスト

にそのファイル名が加わります。複数のファイルをまとめてプロジェクトに追加することもできます。ファイルをそれぞれ追加すればよいでしょう。「すべて追加」ボタンをクリックすれば、上側に一覧されている項目がまとめて下側に登録されます。追加したいファイルを指定すれば、「終了」ボタンをクリックします。すると、指定したファイルがプロジェクトに登録されます。

なお、Finder 上に表示されているソースファイルなどを、Visual Cafe のプロジェクトウインドウ上にドラッグ&ドロップしても、そのファイルを登録することができます。

プロジェクトからファイルを取り除くには、プロジェクトのウインドウでその項目を選択して、「編集」メニューの「クリア」を選択するか、あるいは拡張キーボードなら clear ボタンをクリックすれば良いでしょう。

コンパイルとアプリケーションの作成

プロジェクトに登録したソースファイルは、まずコンパイルします。プロジェクトのウインドウにはタブがあって、いくつかの表示形態を選べますが、ソースプログラムを直接処理したいときには「ファイル」のタブを選択しておくと便利です。その状態でファイル名の項目をダブルクリックすると、そのファイルが開いて編集できるようになります。

図 4-20　プロジェクトに登録したファイルを見る

プログラムファイルのコンパイルは、「プロジェクト」メニューの「コンパイル」（Command+K）でできます。プログラムソースファイルを開いてアクティブな状態にしておき、メニュー選択あるいはキー操作します。あるいはプロジェクトウインドウで項目を選択してコンパイルをかけることもできます。

コンパイルを行うとエラーがなければ、プロジェクトウインドウは各項目の右側に

ついているチェックマークがなくなります。

♦起動するクラスの確認

ソースファイルを作成し、コンパイルができれば実行のための準備にかかります。「プロジェクト」メニューの「オプション」（Command+;）を選択して、オプション設定のダイアログボックスを表示します。いくつかタブがありますが、まず「プロジェクト」のタブのページを設定します。

図 4-21　オプション設定の「プロジェクト」

このページにはいろいろな設定がありますが、まず、アプリケーションを作成するので「プログラム型」のラジオボタンで「アプリケーション」が選択されているのを確認してください。そして、「main()クラス」のポップアップメニューで、アプリケーションを起動時に実行する main メソッドが定義されているクラスを選択します。ここはコンパイルしたソースに含まれるクラスのうち、main メソッドがあるものが自動的に設定されています。main メソッドが存在するクラスが 1 つだけなら、そのクラスが自動的に設定されているので、特に変更はないでしょう。

「main()への引数」は、実行時に main の引数である arg に引き渡されるパラメータの内容を定義することができます。なお、Visual Cafe では実行時にシステムプロパティの追加はできません。

♦作成アプリケーションの設定とリソースの組み込み

次に作成するアプリケーションに関する設定を行います。同じくプロジェクトのオプション設定のダイアログボックスにある「アウトプット」のタブのページで設定を行います。次の図は設定を完了した状態です。

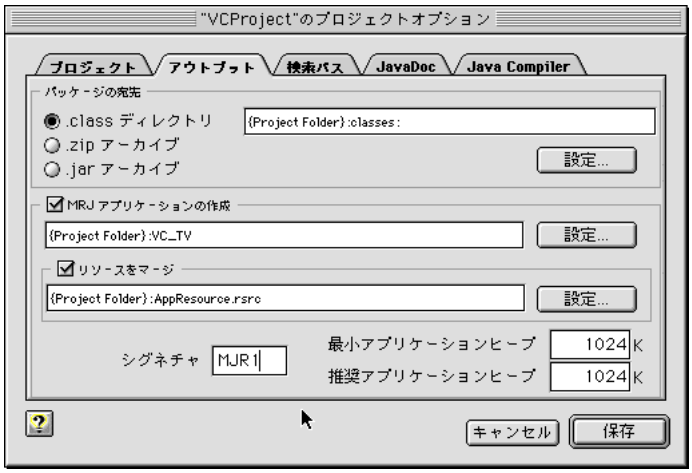

図 4-22　オプション設定の「アウトプット」

まず、「パッケージの編集」については、アプリケーションのもとになるクラスファイルの作成方法を指定するものですので、特にどれかになっている必要はありません。通常は「class ディレクトリ」として、指定したフォルダにクラスごとにファイルに保存しますが、Mac OS は 31 バイトまでのファイル名しか受け入れられません。そのため、たとえば内部クラスを利用した場合などに、31 文字を超えるファイル名になることがあります。その場合は、zip アーカイブや jar アーカイブを選択して、コンパイル結果をアーカイブファイルに保存する必要があります。

そして、「MRJ アプリケーションの作成」というチェックボックスをオンにしておきます。これで MRJ 向けのアプリケーションを作成します。そのすぐ下にあるテキストボックスは、作成するアプリケーションのファイルのパスを示しています。パスの中にある {Project Folder} はプロジェクトファイルが存在するフォルダを示しています。パスをキータイプするのではなく、「設定」ボタンをクリックして、ファイルや保存フォルダを指定します。通常はプロジェクトのあるフォルダと相対的な指定になります。

そして、作成するアプリケーションのクリエーターを「シグネチャ」のテキストボックスにキータイプします。また、ヒープサイズについても適当に設定しておきます。

もし、作成するアプリケーションにリソースを追加するのであれば、まず、そのリソースを ResEdit などを利用して作成しておきます。方法は、4.1 節で説明をしてあります。リソースを用意すると、前の図のオプション設定ダイアログボックスにある「アウトプット」のタブで「リソースをマージ」のチェックボックスをオンにします。そして、「設定」ボタンをクリックして、次のようなダイアログボックスでリソースフ

ァイルを選択します。リソースファイルは、ファイル名の末尾を.rsrc にしておくのが良いでしょう。リソースファイルを選択すると、「リソースをマージ」のチェックボックスの下にあるテキストボックスに、リソースファイルへのパスが設定されます。

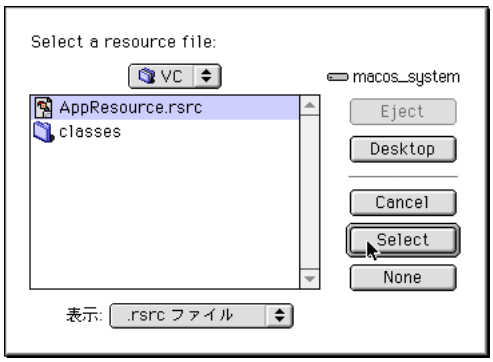

図 4-23　リソースファイルを選択する

アプリケーション化の設定ポイント

こうして必要な設定を行うと、「プロジェクト」メニューの「アプリケーションクラスのビルド」（Command+U）で、アプリケーションのファイルを作成します。なお、Command+R などで実行をした時には、プロジェクト名に.run という文字が付加されたファイル名のアプリケーションが作成され、そのアプリケーションが起動します。ただし、この実行時のアプリケーションにもプロジェクトのオプション設定は適用され、たとえばリソースなどがマージされるので、同じように Mac OS 上のアプリケーションとして利用できます。

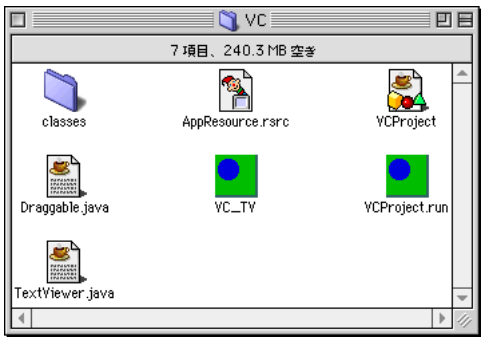

図 4-24　作成したアプリケーションと実行時に作られるアプリケーション

ここで Visual Cafe を使って作成したアプリケーションは、クラスファイルなど実

行に必要なものが 1 つのファイルにまとめられている形式のもので、アプリケーションファイルだけを別のフォルダに移動したり、あるいは MRJ が稼動している別のマシンに移動してそのままダブルクリックして実行できるわけです。このアプリケーションは、JBindery での VFS の機能を利用して 1 つのファイルにまとめたものに対応します。Visual Cafe でのアプリケーション作成中のメッセージを見ると、実行クラスの位置として$VFS を付加したクラスパスを認識しているようです。

Droplet によるアプリケーション作成

Visual Cafe でプロジェクトを用意するときに、Droplet Project を選択すると、Finder でのドラッグ&ドロップを受け付けるアプリケーションのひながたとなるプロジェクトを作成します。作成されるプロジェクトには 2 つのソースファイルが作られますが、1 つは About ダイアログを定義するもので、重要なのはもう 1 つの方です。初期状態では、Frame1 という名前になっていて、ソースファイル名は Frame1.java です。このソースで定義されている Frame1 クラスは、ウインドウである Frame クラスを拡張したものです。この中に main メソッドが定義されています。また、この Frame1 自体に、Required AppleEvent を受け付ける機能が組み込まれています。ソースの概略を示すと次のようになっています（ソースは一部を省略しています。また、日本語のコメントはもとのソースにはなく付け加えたものです）。

```
import com.apple.mrj.MRJApplicationUtils;
import com.apple.mrj.MRJOpenDocumentHandler;
import com.apple.mrj.MRJQuitHandler;
import com.apple.mrj.MRJAboutHandler;

public class Frame1 extends java.awt.Frame
    implements MRJOpenDocumentHandler, MRJQuitHandler, MRJAboutHandler
{
    void userHandleFile(File file)
    {
        if(file != null)
        {
            //Open イベントの時の処理をここに記述する.

        }
    }

    void doAbout()
    {
        //About イベントの時の処理
        //既定値では、最初から定義されている AboutDialog クラスのダイアログを表示
    }
```

```
    void doOnQuit()
    {
        //Quit イベントの時の処理を記述する

        //Exit with success.
        System.exit(0);
    }

    public Frame1()
    {
        //イベントハンドラの登録を行う
        MRJApplicationUtils.registerOpenDocumentHandler(this);
        MRJApplicationUtils.registerQuitHandler(this);
        MRJApplicationUtils.registerAboutHandler(this);

        //ウインドウの初期化やファイルを開くダイアログボックスの用意
        //メニューの構築など（省略）
    }

    static public void main(String args[])
    {
        Frame1 frame1 = new Frame1();
        frame1.show();
    }

    //MRJ Interface
    public void handleOpenFile(File file)
    {
        userHandleFile(file);
    }

    public void handleQuit()
    {
        doOnQuit();
    }

    public void handleAbout()
    {
        doAbout();
    }

    //メニューやダイアログボックスのメンバ定義（省略）

    //メニュー選択により、ファイルを開いたり、アプリケーションの終了
    //      をできるようにするためのイベント処理（省略）
}
```

このプロジェクトを利用すれば、ここからプログラムを追加することで、Mac OS 向けのアプリケーションが作成できるということになります。リソースについては必要に応じて ResEdit で作成して追加すれば良いでしょう。About ダイアログについて

もひながたになるものが定義されているので、そこに表示したいものを追加すればいいわけです。

HandleOpenFile メソッドは定義されていますが、そこからさらに userHandleFile メソッドを呼び出しています。このメソッドはクラス定義の冒頭にありますが、実際にはこちらのメソッドに処理を加えればよいということになります。Quit や About についても、イベントによって呼び出されるメソッドから、さらにクラスの最初に定義しているメソッドを呼び出しています。なお、Print イベントに対応するメソッドは定義されていません。

このプロジェクトを利用すれば、確かにある程度はプログラム作業を軽減はできますが、できあがるのは基本的なレベルのプログラムです。慣れないうちだとこうしたプロジェクトでも便利ですが、MRJToolkit の機能を理解してしまえば、これくらいは何でもないと感じるかも知れません。知っておくと便利ですが、どんな場合でも便利なものとは言い切れないと考えられます。

Visual Cafe Ver.2.0 と MRJ 2.1EA3

MRJ 2.1EA3 をインストールした状態では、Visual Cafe でのデバッグ機能は使えません。デバッグを有効にしても、デバッグには入れません。これについては将来のリリースで修正されるとドキュメントに記載されています。

また、EA3 を使っていると、Visual Cafe のアプリケーションを終了する時に、フリーズします。ただし、MacsBug をインストールしてあれば、es コマンドで復旧することができるようです。

4-3 Code Warrior を使ったアプリケーション作成

Code Warrior Pro4 でアプリケーションを作成してみます。Code Warrior は MRJ だけでなく、Metrowerks 社の Java VM を使った開発も可能ですが、Pro4 からは MRJ 環境での開発を中心的なターゲットになるように機能がかなり修正されました。MRJ で機能させるアプリケーション作成に絞って紹介しましょう。

プロジェクトの用意

Code Warrior で Java 対応のアプリケーションを作る時は、アプリケーション作成用に準備されているプロジェクトを利用します。まず、「ファイル」メニューの「新規プロジェクト」（Command+shift+N）を選択すると次のようなダイアログボックスが表示され、作成するプロジェクトのひながたを選択することができます。ここで、Java の項目を展開し、その中にある Java Application を選択します。

図 4-25 Standalone のひながたを利用する

すると、プロジェクトなどを保存しておくフォルダをどこに作るかを指定するダイアログボックスが表示されます。CodeWarrior Pro4 ではナビゲーションサービスに対応したダイアログボックスが表示されます。以下のように、フォルダを作るフォルダに合わせるか指定をして「保存」ボタンをクリックします。ここでは「CW4」という名前のフォルダが「ch4_DraggableApp」フォルダに作成され、そのフォルダ内に CW4 というファイル名のプロジェクトが作られます。プロジェクトがダイアログボックス

で表示されているフォルダに作成されるのではありません。

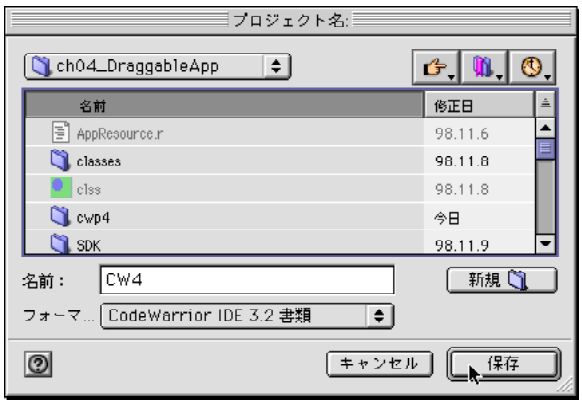

図 4-26　保存するフォルダを指定する

そうすると、プロジェクトが開きますが、簡単なアプリケーションのソースファイルである TrivialApplication.java と、いくつかのライブラリが最初から組み込まれています。ただし、初期状態では必要な設定は完了されておらず、MRJ 環境で実行できるアプリケーションがすぐにできるというわけではありません。

ソースファイルの作成

Code Warrior 上で新たにプログラムソースを作成するのであれば、TrivialApplication.java の内容を書き換えてもかまいませんが、通常は「ファイル」メニューの「新規」（Command+N）を選択して、新たにドキュメントウインドウを開き、適当なファイル名で保存します。そして、そのウインドウにプログラムを作成しますが、「プロジェクト」メニューの「ウインドウを追加」を選択すると、アクティブなウインドウのドキュメントをプロジェクトに追加することができます。

もし、既存のソースファイルがあるのなら、そのファイルをプロジェクトに登録します。プロジェクトのウインドウがアクティブな状態で「プロジェクト」メニューの「ファイルを追加」を選択してダイアログボックスでファイルを追加するか、あるいは、Finder 上のファイルをプロジェクトのウインドウ内にドラッグ&ドロップします。ドラッグ&ドロップしたときには、プロジェクト内のどこに割り込むかが分かるような表示がされるので、それを参照しながら保存したいフォルダのところにドラッグ&ドロップすればいいでしょう。もちろん、いったん登録してから、任意のフォルダに分類してもかまいません。ここでは、Java のソースと、最終的に作成するプログラムに含めたいリソースファイルもまとめてプロジェクトに登録しました。リソースファ

イルは、MRJ SDK での開発で説明したものと同じもので、主に、テキストファイルを Finder 上で認識できるようにする BNDL リソースの定義と、アプリケーションファイルのアイコンの定義が含まれています。

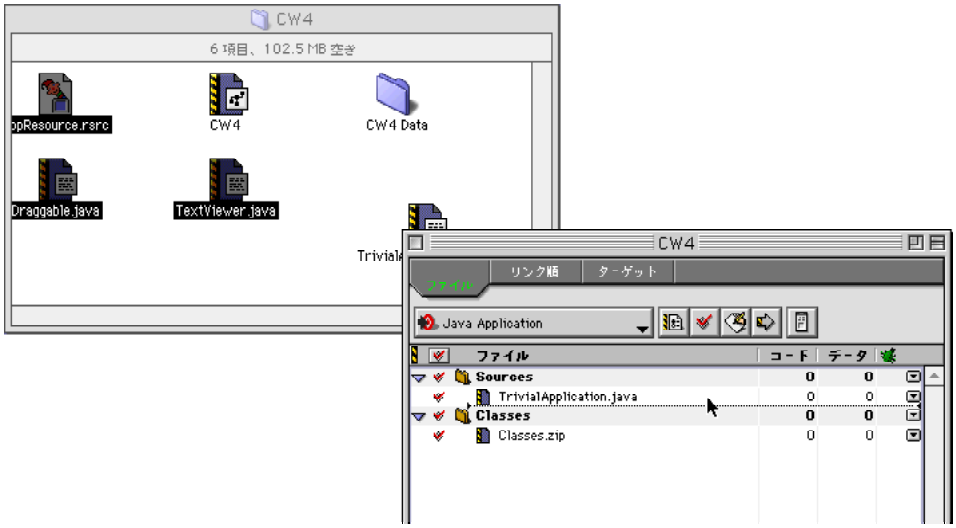

図 4-27　ソースファイルをドラッグ&ドロップで登録する

♦必要なライブラリとクラスファイルの追加

プロジェクトのひながたである Java Application には、最初からライブラリが 1 つだけ含まれています。その classes.zip は、JDK で提供されている基本ライブラリが含まれているファイルで、おおもとのファイルは、Metrowerks:Metrowerks CodeWarrior:Java Support:Libraries:classes.zip にあるものです。AWT などの Java の基本クラスを利用している場合、コンパイル時にはこのファイルに含まれるクラスを参照します。

Java の基本クラスを使うだけのアプリケーションならこれだけでもかまいませんが、Open イベントに対応するなど、MRJ 独自の機能を使っているような場合では、このままの状態ではライブラリが不足します。つまり、com.apple. MRJOpenDocumentHandler などのクラスの定義がプロジェクト内に存在しないので、たとえば import 文のところでエラーが出ます。これら MRJ が提供するクラスの定義もコンパイル時に参照できるようにするために、システムフォルダの「機能拡張」フォルダにある MRJLibraries:MRJClasses:MRJClasses.zip というパスのファイルもプロジェクトに登録しなければなりません。次の図は、プロジェクトのウインドウがアクティブな状態で「プロジェクト」メニューの「ファイルを追加」を選択し、前述のフォルダに移動し

て、MRJClasses.zip というファイルを追加するところです。

図 4-28　MRJClasses.zip というファイルもプロジェクトに追加する

ファイルを指定して「Add」ボタンをクリックすると、プロジェクトに MRJClasses.zip というファイルが追加されます。そのとき、クラスの検索パスに、MRJClasses フォルダが追加されたというメッセージが出てきます。もし、プロジェクトを別の Macintosh に移動したときなどは、この MRJClasses.zip を参照するパスを書き換える必要が出てくるかも知れません。そのような場合には、検索パスをあらかじめ、システム相対パスになるように、「編集」メニューの「Java Application の設定」の「アクセスパス」の項目で設定を変更しておくと、利用する Macintosh を移動しても、確実に MRJClasses.zip を参照するでしょう。

こうして、プロジェクトに登録されたファイルは次の図のようになっています。TrivialApplication.java は不要なので、プロジェクトからは取り除きました。Sources フォルダの内容はアプリケーション次第ですが、Libraries フォルダは図のようになっているのが一般的でしょう。

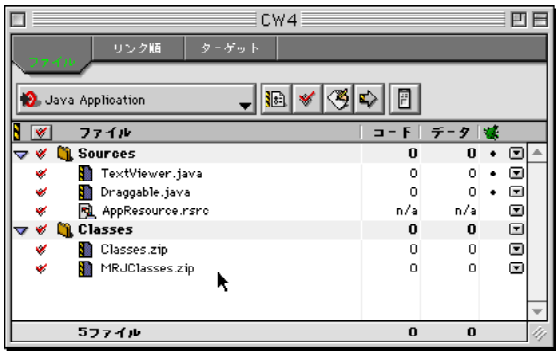

図 4-29　必要なファイルを追加したプロジェクト

アプリケーションの作成に必要な設定

次に、実際にアプリケーション作成に必要な設定を行います。プロジェクトには Java Application という 1 つのターゲットだけが存在するので、それに対して行います。以下の作業は、「編集」メニューの「Java Application の設定」を選択し、設定ダイアログボックスを表示して行います。

まず左のリストから「ターゲット設定」を選択します。ここでは、以下のように、「ポストリンカ」のところで「Java MacOS Post Linker」を選択します。後の設定はそのままでかまいませんが、必要ならターゲット名を変えてもよいでしょう。

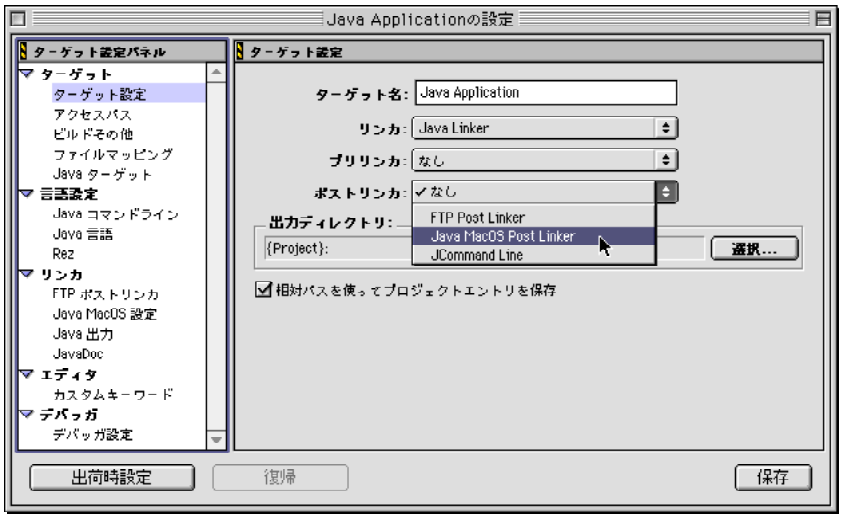

図 4-30　ポストリンカを設定する

次にダイアログボックスの左のリストで「Java ターゲット」を選択します。ここで、

「ターゲットタイプ」が「アプリケーション」になっているのを確認して、「Main クラス」の右のテキストボックスに、main メソッドが定義されいてるクラス名を正確が、キータイプします。そして、「バーチャルマシン」には使用する VM を選択します。最初から AppleMRJ が選択されていますのでそのままでかまいません。

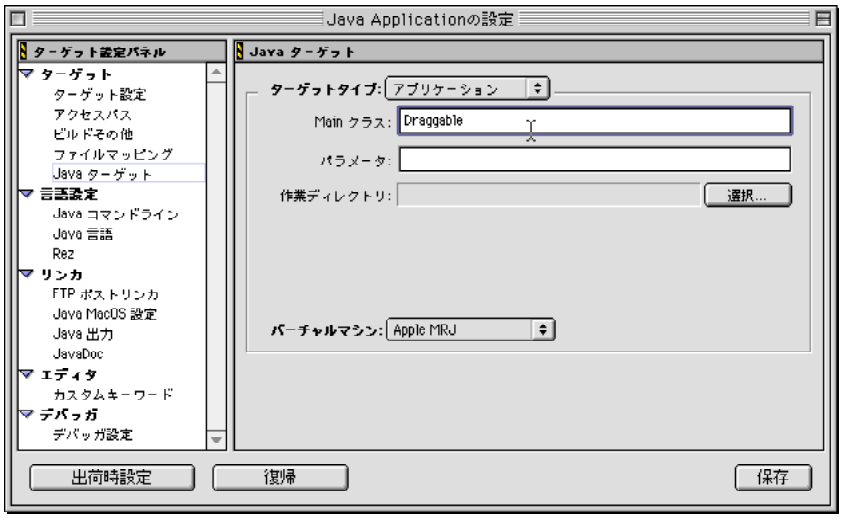

図 4-31　Main クラスを指定する

さらに「Java MacOS の設定」の項目を左側で選択します。「Mac OS Java の出力タイプ」が JBindery になっていることを確認します。そして、下の方にある「クリエータ」では、作成したアプリケーションに付けるクリエイターをキータイプします。「出力ファイル名」はアプリケーションファイルの名前をやはりキータイプします。ヒープサイズは適当に設定しておくとよいでしょう。

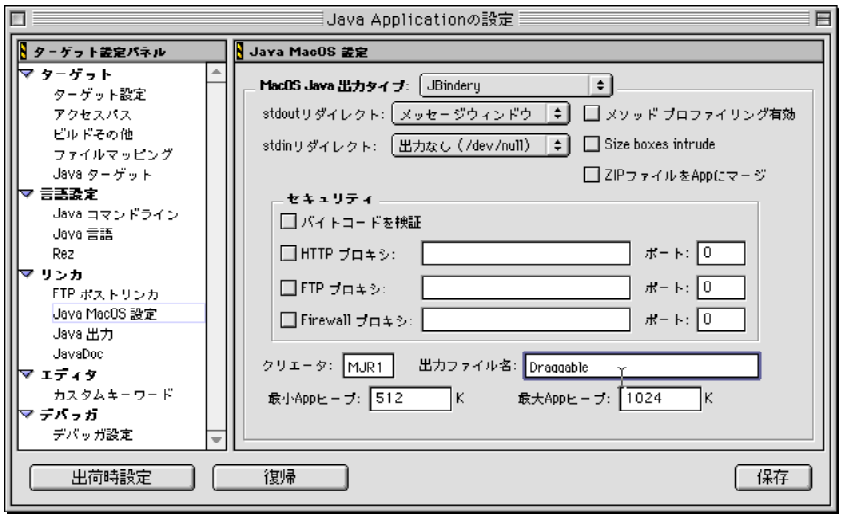

図 4-32　作成されるアプリケーションファイルの設定

以上の設定を行い、「保存」ボタンをクリックしておけば、設定が保存されます。

リソースをアプリケーションにマージさせる

以上でアプリケーション化に必要な設定は完了しますが、さらに、リソースファイルを最終的なアプリケーションに含めるための設定を行う必要があります。リソースファイルは ResEdit で作成したもの、つまりファイルタイプが rsrc であるものとします。リソースファイルは前述のように、プロジェクトに含めておきます。

そして、「編集」メニューの「Java Application の設定」を選択し、左側の「ファイルマッピング」を選択します。そこにはすでに一覧の中にファイルタイプが rsrc に対応する設定があるはずなので、その項目を選択します。そして、「コンパイラ」のポップアップメニューから「JAR Importer」を選択し、「変更」ボタンをクリックします。つまり、リソースについては何もコンパイラが設定されていない状態ではなく、JAR Importer で処理されるようにしておきます。もちろん、設定を変更して「保存」ボタンをクリックしておきます。

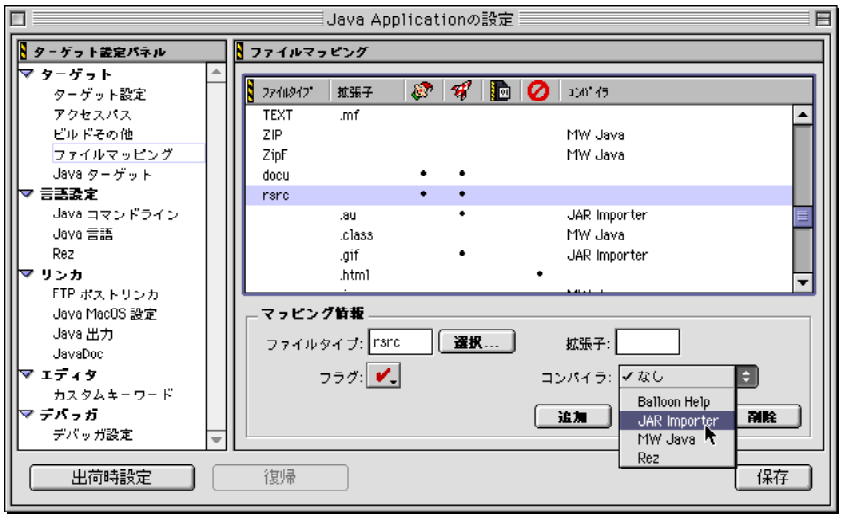

図 4-33　リソースファイルに対して JAR Importer を機能させる

次にプロジェクトウインドウをアクティブにして、リソースファイルに対する設定を変更します。プロジェクトのウインドウで、追加したリソースファイル（以上の例では、AppResource.rsrc）の項目を選択して、「ウインドウ」メニューから「プロジェクトインスペクタ」を選択します。次の図のようなダイアログボックスが表示される

ので、「オプション」にある「生成ファイルにマージ」にチェックを入れておきます。そして、「保存」ボタンをクリックします。そしてウインドウのクローズボックスをクリックしてウインドウを閉じておけばよいでしょう。

図 4-34　作成アプリケーションにマージするようにしておく

以上のように行えば、プロジェクトに追加したリソースファイルのリソースをそのまま生成したアプリケーションにもコピーします。

コンパイルとアプリケーションの生成

こうして「プロジェクト」メニューから「メイク」（Command+M）を選択すると、アプリケーションが作成されます。以下の図のように、Java のソースをコンパイルした結果は、AppClasses.jar というファイルにパッケージされて保存されています。このファイルについての設定は「Java Application の設定」の「Java 出力」の項目で、ファイル名やパスなどを指定することができます。そして、Draggable というアプリケーションファイルが作成されていて、アイコンが割り当てられています。このアイコンにテキストファイルをドラッグ&ドロップすると、アプリケーションが起動し、テキストファイルを開きます。

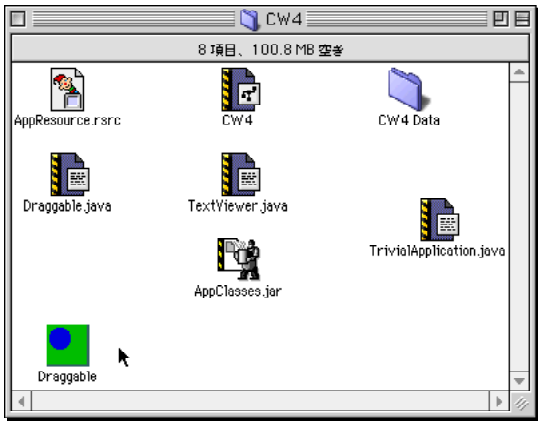

図 4-35　作成したアプリケーション

通常は、ここでの Draggable と AppClasses.jar がセットになっていなければなりません。たとえば、AppClasses.jar を削除してしまうと、このアプリケーションの場合は main の存在する Draggable というクラスを見つけられなくなり、実行時のエラーになります。ただ、AppClasses.jar はサブクラスに移動するくらいであれば、存在を認識するようです。

♦コンパイル結果をアプリケーションに含める

配付して使うなど、実用アプリケーションの場合は、「Java Application の設定」のダイアログボックスにある「Java MacOS 設定」にある「ZIP ファイルを APP にマージ」のチェックボックスをオンにしておきます。そうすれば、この場合では、AppClasses.jar の内容を含んだ Draggable アプリケーションを作成します。つまり、Draggable というアプリケーションファイルに、コンパイルした Java のバイトコードがすべて含まれるので、MRJ がインストールされているシステムであれば実行できるアプリケーションが作成できるということになります。チェックボックス名は ZIP ファイルとなっていますが、JAR ファイルでもかまわないようです。JAR は中身的には ZIP ファイルと基本的に同様ですし、チェックボックスは「アーカイブファイル」という意味で ZIP ファイルと呼んでいるのだと思われます。

4-4　アプリケーション作成環境を比較する

原稿を執筆している 98 年 11 月現在、MRJ 2.1 EA3 が利用できる状態です。VM や開発ツールなどで入手できる最新版を考慮して、どのような環境がアプリケーション作成に適しているかを検討してみましょう

開発環境の比較

まず、コンパイラを比較すると、いちばん問題がないのが Visual Cafe ということになります。MRJ SDK の javac は、¥記号をエスケープ文字として利用できないために、文字列の指定が苦しくなるという事情があります。Code Warrior の Java コンパイラでは、Pro3 までは文字列リテラルに漢字などの 2 バイトコードを使えませんでしたが、Pro4 からは対応していて、ソースコード中に漢字を含む 2 バイト文字を記述できるようになりました。

プログラム作成エディタでは、Visual Cafe も Code Warrior も、キーワードをカラー表示したり、日本語の文字列の扱いも正しくでき、その点では同等です。クラスブラウザが使えるなど、基本的な環境も整っています。一方、MRJ SDK ではそうしたプログラム編集環境はいっさいありません。そのため、キーワードのカラー表示が可能な BBEdit を使ってプログラムを作成し、コンパイルを javac で行うというような使い方がされています。

デバッグについては、MRJ 2.1EA3 の段階では、Code Warrior しかできません。Visual Cafe ではデバッグができませんし、MRJ SDK にはソースコードのデバッガがありません。

RAD 環境が必須となると、Visual Cafe が第一の選択肢になると思われますが、Macintosh 版がバージョンアップされない現状で、しかも Swing に対応していないことを考えれば、デバッグに不安定要素があることなども含めて Java2（JDK1.2）世代の開発ツールとしては不向きになっていると言わざるを得ません。

一方、CodeWarrior は Pro4 になって MRJ 環境にかなりマッチしたつくりになり、また 2 バイト文字のソースへの入力もできるので、現状ではいちばん有力な開発ツールとなるでしょう。また、Java だけの開発環境としても別製品として安価にリリースされる予定もあり、Macintosh 環境では、C/C++に続いて CodeWarrior が Java の標準ツールになったと言ってもよいでしょう。ただし、RAD 環境が整備されていませんが、

Pro5 などのタイミングで RAD ツールを新たにリリースする予定もあり、着実に開発環境として整えてきています。

アプリケーション化の機能の比較

アプリケーション化の機能で、いちばんバリエーションがあるのはやはり、MRJ SDK にある JBindery ということになるでしょう。Visual Cafe では 1 つのファイルにまとまったものしか作ることができませんが、実用的な意味ではあまり問題にならないかと思われます。CodeWarrior は、VFS の設定はできないものの、コンパイル結果を 1 つのファイルに含めることができるので、やはり実用上は十分だと言えるでしょう。

Code Warrior では、Java VM として MRJ だけでなく、Metrowerks 社の VM も選択できます。ただし、将来的なことを考えれば、Java VM 自体は MRJ を利用することで集約されると思われます。その意味では、Metrowerks 社の VM を使うことで、たとえば JDK のバージョンアップに対応できないことや、MRJ の機能が使えないなどのデメリットが出てくることが予想されます。

LOCUS Report

第5章　ウインドウやメニューの取り扱い

● ウインドウやメニューの処理は、Java の標準ライブラリに組み込まれた機能を使ってプログラミングします。基本的には Macintosh だからといって特別な作業は必要ありません。

● この章では、文書ごとにウインドウを開く形式のアプリケーションを試作しました。AWT のウインドウやメニューを利用しています。

● メニューとウインドウが分離している Mac OS 独特の事情に対処する 1 つの方法を示します。

5-1 AWT のウインドウ機能

AWT のウインドウ作成方法について簡単にまとめておきます。Mac OS **の** Toolbox **の場合はウインドウを独立したものと見ますが、**AWT **では、**Frame **というクラスを拡張して独自の機能を持ったウインドウを作成するというのが基本です。**

Frame クラスを継承する

初期の Java ではアプレットを作ることが多かったので、ウインドウについてはあまり中心的な話題に上らない傾向がありました。しかしながら、アプリケーション環境では、ウインドウは必須です。そこで、Java の基本パッケージの中でグラフィックス処理を行う AWT（Abstruct Windowing Toolkit）では、Mac OS や Windows などの GUI の OS で機能するマルチウインドウ処理を利用できるようになっています。そのために、java.awt.Frame というクラスが定義されています。

まず、Frame クラスは、いくつかのクラスを継承しています。順番にたどれば、Object →Component→Container→Window→Frame という順序になっています。重要なのは Component のサブクラスであることを忘れないようにするというところでしょう。ウインドウのサイズを設定するなどの基本的な機能は、Component クラスのメソッドを利用します。また、Window というクラスがありますが、これは、通常のウインドウの Frame クラスと、ダイアログボックスを表示するための Dialog クラスのスーパークラスとなっているため、通常の文書ウインドウ的な表示では、Frame クラスを利用することになります。

この章も含めて、いくつかの章で示してきたサンプルは、ウインドウにテキストファイルの内容を表示する、テキストビューア機能を組み込んでいます。こうした機能を組み込む場合、方針として、Frame クラスを拡張します。Frame は単なるウインドウだと思って下さい。そのウインドウに新たに、テキストを表示するという機能を付加するのです。そして、「テキストを表示するウインドウ」という新しいクラスを定義し、実行時にはそのインスタンスを生成することで、実際にウインドウを表示します。ここでは、テキストビューアの TextViewer というクラスを定義して、このクラスにテキストをウインドウ表示する機能を組み込むことにします。まず、大枠として

は、次のようなクラス定義を行うことになります。

```
import java.awt.*;
import java.io.*;

class TextViewer extends Frame
{
    public TextViewer(File filename)  {
        //      コンストラクタ
    }
}
```

この、TextViewer クラスのインスタンスを生成すると、画面にウインドウを表示する準備が整います。

代表的なウインドウ処理

ウインドウを表示したり、隠したり、サイズを変えるなどの基本的なメソッドの代表的なものを以下の表にまとめました。いずれも、どこかのクラスで定義されたものですが、Frame を拡張したクラスで必ず使えます。なお、この表はすべてのメソッドを紹介したものではありませんので、詳細は API のドキュメントを御覧ください。

表 5-1　Frame **クラスで利用できるウインドウ処理の代表的なメソッド**

戻り値	メソッド	機能
void	hide()	ウインドウを隠す
void	show()	ウインドウを表示する
void	dispose()	ウインドウを閉じる
Rectangle	getBounds()	ウインドウの左上の位置、幅と高さを得る
void	setBounds(int, int, int, int)	ウインドウの左上の位置、幅と高さを変更する
void	setBounds(Rectangle)	ウインドウの左上の位置、幅と高さを変更する
Point	getLocation()	ウインドウの左上の位置を得る
Point	getLocationOnScreen()	ウインドウの左上の位置を画面上の座標で得る
void	setLocation(int, int)	ウインドウの左上の位置変更する
void	setLocation(Point)	ウインドウの左上の位置変更する
Dimension	getSize()	ウインドウの幅と高さを得る
void	setSize(int, int)	ウインドウの幅と高さを変更する
void	setSize(Dimension)	ウインドウの幅と高さを変更する
String	getTitle()	ウインドウのタイトルの文字列を得る
void	setTitle(String)	ウインドウのタイトルの文字列を設定する
boolean	isResizable()	ウインドウがサイズが変更可能かどうか
void	setResizable(boolean)	ウインドウのサイズが変更可能かどうかを設定
void	toBack()	ウインドウを背後に回す
void	toFront()	ウインドウをいちばん手前に持ってくる

Toolkit	getToolkit()	ツールキットを得る
void	add(Component…)	コンポーネントを追加する
void	setLayout(LayoutManager)	レイアウト機能を設定する

5-4 節のサンプルプログラムのうち、TextViewer.java を参照してください。ここのコンストラクタの最初の部分で、ウインドウのタイトルを設定したり、あるいはウインドウのサイズを変更したり、実際に表示するなどの処理が組み込まれています。コンストラクタ内では、メソッドだけの記述で、生成した TextViewer オブジェクトに対するメソッドとして処理されます。

イベントリスナを組み込む

ウインドウとして表示した場合、通常は何もしなくても、たとえばサイズを変更できたり、タイトルバーの部分をドラッグして移動したりということができます。つまり、OS レベルでサポートされているウインドウ処理の多くは、Frame クラスで機能が組み込まれているため、それを継承したウインドウ表示クラスでは、そのまま基本的な機能を利用できるのです。

さらに、以下のようなメソッドを利用して、イベントリスナを組み込むことができます。組み込み可能なリスナのクラスは、各メソッドの引数に記述されています。Frame はいろいろなクラスを継承していることもあって、たくさんの種類のリスナを利用できますが、必ずしもすべてを利用する必要はありません。何もしなくても使える機能に新たに機能を加えた場合にのみ、こうしたイベント処理が必要になります。

```
void    addWindowListener(WindowListener)
void    addContainerListener(ContainerListener)
void    addComponentListener(ComponentListener)
void    addFocusListener(FocusListener)
void    addKeyListener(KeyListener)
void    addMouseListener(MouseListener)
void    addMouseMotionListener(MouseMotionListener)
```

TextViewer では、WindowListener と ComponentListener を利用します。まず、ウインドウの処理は基本処理は何もしなくても自動的に組み込まれますが、クローズボックスのクリックによってウインドウを閉じるという処理については、Frame クラスでは機能として含まれていません。クローズボックスをクリックしたときに、dispose メソッドを呼び出して、ウインドウをクローズしなければなりません。

♦クローズボックスの処理

クローズボックスのクリックは、WindowEvent（ウインドウイベント）として発生

されるため、そのイベントを受け取る WindowListener の機能をクラスに組み込む必要があります。そのために、Frame を継承した TextViewer クラスの最初の部分で、implements WindowListener という記述を追加します。そして、TextViewer のコンストラクタで、作成したオブジェクト（this で参照できる）を、addWindowListener メソッドを使ってイベントの受け取りオブジェクトとして登録をしておきます。

WindowListener をインプリメントしたクラスでは、以下に示すように、WindowClosing などいくつかの abstruct 宣言されたメソッドは必ず定義します。特に何も処理がなくてもメソッドは用意しておかないとエラーになります。このうち、ウインドウのクローズボックスをクリックしたときには、windowClosing というメソッドが呼び出されるので、そこで dispose メソッドを呼び出して、ウインドウを実際に閉じます。

```
public void windowClosing(WindowEvent e)
{
        dispose();  //ウインドウを閉じる
                    //これがないとクローズボックスをクリックしてもウインドウは閉じない
}
public void windowOpened(WindowEvent e){}
public void windowClosed(WindowEvent e){}
public void windowIconified(WindowEvent e){}
public void windowDeiconified(WindowEvent e){}
public void windowActivated(WindowEvent e){}
public void windowDeactivated(WindowEvent e){}
```

♦ウインドウのサイズ変更への対応

TextViewer では、ウインドウの中に、TextArea コンポーネントを埋め込んで、それを使ってテキストを表示します。TextArea コンポーネントは、ウインドウの大きさとぴったり同じにしておいて、ウインドウ全体でテキスト表示ができるようにします。ここで、Frame でレイアウトの機能を使った場合には、必ずしも必要ないのですが、レイアウトを使わないようにしたので、ウインドウのサイズを変化させたときに合わせて、その中の TextArea のサイズも変更します。

ウインドウの大きさを変更した時には、ComponentEvent（コンポーネントイベント）が発生します。まず、コンポーネントイベントを受け取ることができるように、TextViewer の定義の最初の部分に、ComponentListener をインプリメントします。TextViewer のコンストラクタでは、生成した TextViewer のオブジェクトに対して、addComponentListner メソッドを使って、イベントを受け取ることができるようにします。そして、クラス内に以下のようないくつかのメソッドを定義しておきます。このうち、ウインドウのサイズを変えた時には、componentResized メソッドが呼び出され

るので、ここで、ウインドウの大きさと TextArea の大きさの同期を取っています。

```
/*ウインドウのサイズを変更した時に呼び出される*/
public void componentResized(ComponentEvent e)
{
        System.out.println("Window Resized");
        viewText.setSize(this.getSize());
                        //TextArea の大きさをウインドウの変更後のサイズに合わせる
}

public void componentMoved(ComponentEvent e){}
public void componentShown(ComponentEvent e){}
public void componentHidden(ComponentEvent e){}
```

Mac OS のウインドウには、右上に折り畳みボタンや、最大化ボタンがあります。これらのボタンについては特に何もしなくても、Frame クラスの継承クラスではユーザーのクリックを受け付けます。

5-2 AWT のメニュー機能

AWT には、メニューを作成する機能があります。ただし、Windows などの GUI 形式に近く、メニューはウインドウに所属するものとして扱われます。Mac OS では Mac OS の方法でメニューを表示しますが、いくらかの違和感のあるところです。AWT でのメニュー利用方法の基本的なことをまず説明しましょう。

メニュー作成用のクラスを利用

メニューの存在自体は階層的になっているので、オブジェクトとして考えやすいところかもしれません。まず、メニュー項目がいくつか集まって 1 つのメニューを構成します。そのメニューがいくつか集まってメニューバーが構成されています。メニュー項目 1 つ 1 つには、ショートカットもあります。これらは以下のようなクラスを組み合わせて使います。コンストラクタは代表的なもので、ほかにもいくつかパターンはあります。

表 5-2 メニューで利用されるクラス

クラス名	コンストラクタ	オブジェクトの内容
MenuBar	MenuBar()	メニューバー
Menu	Menu(String, boolean)	1 つのメニュー。コンストラクタの 1 つ目の引数にはメニューバー内に表示される文字列を指定。2 つ目の引数はティアオフメニューにするかどうか
MenuItem	MenuItem(String, MenuShortcut)	メニューの中の 1 つの項目。コンストラクタにはメニュー項目名を指定。ショートカットの指定は省略できる
MenuShortcut	MenuShortcut(int, boolean)	ショートカット項目。コンストラクタの 1 つ目の引数はキーコード。2 つ目の引数は Shift を含めるかどうかを指定するが、省略することもできる

こうしたオブジェクトを組み合わせて、メニューバーを作成し、それを Frame オブジェクトに追加します。もちろん、Frame を継承したクラスに付け加えるのが一般的でしょう。

Mac OS では、ウインドウとメニューは独立した存在ですし、実際に Toolbox の機

能としても個別に用意されています。また、メニューはアプリケーションで用意して、それとは別に文書ウインドウを処理するというのが Mac OS 的なやり方です。ところが、AWT では Windows と同様に、1 つのウインドウの中にメニューを追加するという手法を採用しているので、そのように処理をしなければなりません。しかしながら、Mac OS で実行したとき、ウインドウ内にメニューがあるのではなく、通常のメニューバーにメニューが表示されます。そしてウインドウが切り替わるときに、それぞれのウインドウに登録されたメニューバーに逐一切り替わります。こうした状態の不一致がありますが、それをなんとかしのぐ方法は 5-3 で説明します。

Frame にメニューバーを登録するなど処理には以下のようなメソッドが用意されています。以下のメソッドは代表的なもので、他にメニューの途中に項目を追加するような機能などがありますが、それは API のリファレンスを参照してください。

表 5-3　代表的なメニュー処理のメソッド

クラス	戻り値	メソッド	機能
Frame	MenuBar	getMenuBar()	Frame に設定されているメニューバーを得る
Frame	void	setMenuBar(MenuBar)	メニューバーを Frame に追加する
MenuBar	Menu	add(Menu)	引数のメニューをメニューバーに追加する
Menu	MenuItem	add(MenuItem)	引数に指定したメニュー項目を追加する。引数に Menu オブジェクトを指定して、階層メニューも追加できる
Menu	void	addSeparator()	メニューの区切り線を追加する
MenuItem	String	getLabel()	メニュー項目の文字列を得る
MenuItem	void	setEnabled(boolean)	引数によって選択可能、不可能を切り替える
MenuItem	boolean	isEnabled()	選択可能な項目なら True を戻す
MenuItem	void	addActionListener(ActionListener)	メニュー項目を選択したときにイベントを送付するクラスを定義する
MenuBar	void	setHelpMenu(Menu)	引数のメニューをヘルプメニューにする

実際にメニューを作成するプログラムは、5-4 のサンプルプログラムを御覧ください。基本的にはメニュー項目を 1 つ 1 つ生成して、それをメニューに追加し、さらにメニューバーに追加するということを行うため、プログラムは長くなりがちです。

サンプルでは、階層メニューになるような箇所もあります。実際に実行させた結果とプログラムを突き合わせれば、階層メニューを作る参考になるかと思います。

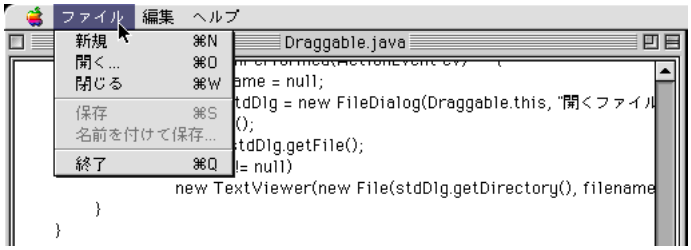

図 5-1　TextViewer で作成した「ファイル」メニュー

図 5-2　「編集」メニューと階層化された「背景色」の項目

イベントリスナを組み込む

メニュー自体はひたすらオブジェクトを生成して作ることになりますが、メニュー項目を選択したときに、それぞれの項目ごとに処理を続けなければなりません。つまり、メニュー選択を受け付けるような仕組みをプログラムに組み込む必要があります。その場合、動作するプログラムの作り方は 1 通りではなく、いろいろなやり方があります。サンプルプログラムでは、メニューが選択されたときに、あるクラスが呼び出されるような作りになっています。たとえば、「ファイル」メニューの「新規」を選択したとき、TextViewer クラスの内部クラスとして定義した NewItemListener クラスが呼び出されるようにします。

メニューの選択も、やはりイベントとして処理されますが、ActionEvent（アクションイベント）としてイベントが発生します。アクションイベントを処理する機能は ActionListener によって提供されるので、NetItemListener クラスでは ActionListener をインプリメントしておきます。そうすると、NewItemListener クラス内の actionPerformed メソッドが呼び出されます。

メニュー項目を作ってメニューバーを構築した後に、各項目に対してリスナの設定を行っています。「新規」の場合には、メニューを選択してアクションイベントが発

生したとき、NewItemListener を呼び出したいので、まずは NewItemListener クラスのインスタンスを生成し、addActionListener でメニュー項目とイベントが送付されるオブジェクトを結び付けてやります。

こうした一連の作業を各メニュー項目に対して行うことになります。プログラム自体は長くなりますが、各メニューでやっていることは基本的には同じです。メニュー選択後の処理はメニュー項目ごとに違っています。

サンプルプログラムでは、「編集」メニューや「背景色」のメニューでは、いくつかのメニュー項目からのアクションイベントを同一のオブジェクトに伝達されるようにしています。たとえば、「カット」でも「コピー」でも、EditItemListener クラスの同一インスタンスが呼び出されるようにしています。従って、EditItemListener の actionPerformed メソッドでは、選択されたメニュー項目ごとにそれぞれ違う処理に分岐する必要があります。どのメニュー項目を選択したかは、引数の ActionEvent オブジェクトから MenuItem を取り出し、getLabel メソッドで選択したメニュー項目名を取り出して判断にかけています。

◆ショートカットの処理

ショートカットはキーボード処理なので、イベントとしては別のものが発生するというのが一般的な考えかもしれませんが、AWT ではショートカットとして登録したキー操作を行うと、それは対応するメニュー項目を選択したときとまったく同様にアクションイベントが発生します。つまり、ショートカットを処理するためのプログラムはわざわざ書かなくても、メニュー選択のクラスさえ定義しておけばそれでショートカットの処理は行われるのです。

ヘルプメニューの取り扱い

Mac OS のアプリケーションでは、必ず「ヘルプ」メニューが登場します。Java のアプリケーションでも、やはりシステムが提供するヘルプメニューが自動的に追加されます。そのヘルプメニューにメニュー項目を追加することもできます。具体例はサンプルをごらんください。

まず、「ヘルプ」メニューの Menu オブジェクトを作り、追加したい項目を MenuItem として追加しておきます。そして、それを MenuBar に追加しますが、それだけだと、メニューバーに「ヘルプ」メニューが 2 つ含まれます。そこで、setHelpMenu メソッドを実行すれば、2 つのヘルプメニューが合体して 1 つになります。

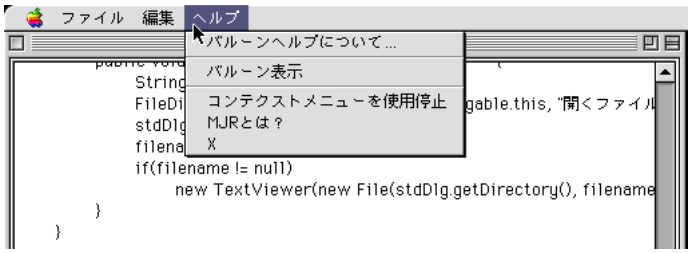

図 5-3　「ヘルプ」メニューに項目を追加したが、ゴミ？も入っている

この setHelpMenu は、Windows でも UNIX でも同様に使えるのですが、OS ごとに微妙にヘルプメニューの出方が違っています。いずれにしても、ヘルプメニューのオブジェクトを用意する点ではどのプラットフォームでも共通です。

MRJ のショートカット組み込み機能

MRJToolkit には、MRJMenuUtils クラスがあり、そこで setMenuItemCmdKey というクラスメソッドが用意されています。これにより、Mac OS 独自のコマンドキーとのキーボードショートカットを付加するという機能が提供されています。

しかしながら、この機能は、AWT にショートカットのクラスが定義されていなかった JDK 1.0 時代のもので、JDK 1.1 では、AWT の MenuShortcut クラスを使えば、Mac OS 上では、Command+C などのキーボードショートカットが使えるようになりました。同じプログラムで、Windows や UNIX では Ctrl+C になるなど、OS 間の違いは、Java VM あるいはライブラリ内で吸収されています。

したがって、現状の Java では、MRJToolkit のメニューのショートカットを使う機能はもはや使う必要がないと言えるでしょう。

5-3 MRJ 環境でのアプリケーション

メニューバーとウインドウが分離した Mac OS で、メニューや複数の文書ウインドウを持つようなアプリケーションを Java で作成する場合の問題点と 1 つの解決策を示します。

ウインドウとメニューが分離した Mac OS アプリケーション

Windows などの大多数の GUI が、ウインドウの中にメニューを持つ形式になっているため、Frame クラスに MenuBar を追加するメソッドが用意されています。一方、Mac OS では、メニューとウインドウは分離されていて、Toolbox レベルでは、メニューバーとウインドウを関連づけるような機能は提供されていません。

Java のアプリケーションのサンプルプログラムなどを実際に使って簡単なアプリケーションを作るのならともかく、文書ウインドウがマルチウインドウで開くようなものを作成すると次のような問題が出ます。ここでは、この章のサンプルプログラムの動きを見て下さい。

サンプルプログラムは、Open イベントなどを受け付ける、いわばアプリケーションの核になるような Draggable というクラスと、ウインドウにテキストを表示する TextViewer というクラスで構成されています。Draggable に main メソッドがあり、基本的には Draggable のコンストラクタはそこにだけあるので、インスタンス化されるのは起動時の 1 度だけです。

一方、TextViewer はファイルを開けばそれだけウインドウが増えて表示されるようにしてあるので、いくつもインスタンスは作られます。

♦文書ウインドウには一般的なメニュー作成手法を使う

メニュー項目は、いくつか例外はありますが、アクティブウインドウに対する処理を行うのが一般的なので、その意味では Frame に所属するという考え方は理にかなっていると言えます。TextViewer クラスの定義では、そのウインドウがアクティブになった時に必要なメニュー項目を定義するということで、これは一般的な Java のプログラムの手法をそのまま利用すれば良いでしょう。サンプルでもそうしています。ただ、Mac OS 的に見れば、1 つのメニューバーをいくつかのウインドウで共有する形式が一般的なので、ウインドウの数だけメニューバーを用意するというのは非効率だ

と思うかもしれません。実際、アクティブウインドウを切り替える時にメニューバーも切り替わるのですが、切り替わりが目で見て分かるくらいでもあります。ただ、Mac OS 向けに 1 つの MenuBar で済ませるということをした場合、Windows などでもアプリケーションを使うとなったときにそのままでは済まないかもしれません。また、そうしたプログラムは作りづらくなっています。ともかく、Frame でウインドウを表示する場合、一般的な手法でメニューを構築することで対処するのが適当だと言えます。

♦ウインドウが 1 つもない状況が生じる

ただし、そうすると、ウインドウが 1 つも開いていないという状況が発生します。もちろん、そのような場合にはアプリケーションを終了するというのが 1 つの方法ですが、Mac OS のアプリケーションでは、一般にはウインドウがなくてもアプリケーションは起動したままになります。たとえば、第 4 章でのサンプルプログラムを起動して、ウインドウをすべて閉じてみて下さい。その場合でも、アプリケーションは起動したままになっています。

こうしたウインドウが 1 つも開いていないときにも、「ファイル」メニューの「新規」「開く」「終了」くらいのメニューが用意されていて、新たにファイルを開いたりということをしたいと考えるところです。しかしながら、メニューは Frame に対して追加できるものなので、「デフォルトのメニュー」的な機能は用意されていません。

Mac OS 的なアプリケーションの試作

そこで、サンプルアプリケーションのような形態の場合、最初に起動される Draggable というクラスで Frame を継承させるようにします。第 4 章のサンプルでは、Draggable は明示的には継承クラスを指定しませんでしたが、この章では、Frame を継承して、起動時にウインドウを表示することにします。そうすれば、TextViewer が存在しないときのメニューをそこで定義できます。

ただし、そのようにすると、今度は Draggable のウインドウが画面に表示されてしまい、邪魔なウインドウが増えてしまいます。そこで、Draggable によるウインドウは、画面の外に追いやってしまって、あたかも見えていないようにします。ウインドウが表示されていないとメニューは見えないので、hide で隠すわけには行きません。そこで、画面の外にウインドウを表示して、メニューだけが見えるようにするというわけです。

サンプルアプリケーションでは、そうした方針で Draggable クラスを定義していま

す。そこでのメニューは、TextViewer クラスほどたくさんの項目は必要ありません。「編集」メニューは不要で、「ファイル」メニューの必要な項目だけを定義しているという次第です。

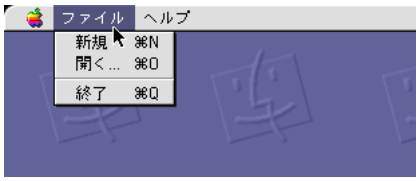

図 5-4　アクティブなウインドウがない時のメニューも表示されるようにした

ただ、こうした処置をしてしまうと、今度は Windows で実行させるときになんらかの工夫が必要になるでしょう。こうした点は別の章で論じることにします。

5-4 サンプルアプリケーション

この章のテーマに沿ったサンプルアプリケーションのソースです。テキストファイルをウインドウで開いて表示するというもので、文書ウインドウをマルチウインドウで扱うようなアプリケーションの基本形とも言えるものです。ファイルは、Draggable.java と TextViewer.java の 2 つですが、アプリケーション化には、BNDL リソースとアイコンを定義したリソースファイルが必要です。リソースファイルは第 4 章のものをそのまま利用します。

Draggable.java

```
import com.apple.mrj.*;
import java.io.*;
import java.util.*;
import java.awt.*;
import java.awt.event.*;

public class Draggable extends Frame
    implements      MRJOpenDocumentHandler,
                    MRJPrintDocumentHandler,
                    MRJQuitHandler,
                    MRJAboutHandler
{
    static public void main(String arg[])      //起動時に実行されるメソッド
    {
        new Draggable();      //このクラスを新たに生成する
    }

    public Draggable()      //コンストラクタ
    {
        /*基本 AppleEvent が、このクラスに伝達されるように定義する*/
        MRJApplicationUtils.registerOpenDocumentHandler(this);
        MRJApplicationUtils.registerPrintDocumentHandler(this);
        MRJApplicationUtils.registerQuitHandler(this);
        MRJApplicationUtils.registerAboutHandler(this);

        setupMenuBar();//メニューバーの設定

        setBounds(-1000,-100,10,10);      //ウインドウの位置を画面の外側に出す。
            //つまり非表示にしたいのだが、hide()とするとメニューまで消えるので、
            //こういう方針にした
        show();     //ウインドウを表示するが、実際には画面には見えない
    }
```

```
public void handleOpenFile(File filename)       //Open イベントが Finder からやってきたとき
{
        System.out.println("Dragged "+filename.toString());
        TextViewer newViewer = new TextViewer(filename);
                                                //開くファイルを TextViewer で表示する
}

public void handlePrintFile(File filename)      //Print イベントが Finder からやってきたとき
{
        System.out.println("Print Event "+filename.toString());
        TextViewer newViewer = new TextViewer(filename);
                                                //開くファイルを TextViewer で表示する
        PrintJob pj = newViewer.getToolkit().getPrintJob(newViewer,"A",null);
                                        //印刷設定のダイアログボックスが表示される
        Graphics pg = pj.getGraphics();     //印刷時のグラフィックス環境を取得
        if(pg != null)        {
                newViewer.printAll(pg);         //印刷メソッドを呼び出す
                pg.dispose();       //グラフィックス領域の破棄により、実際に印刷が始まる
        }
        pj.end();
}

public void handleQuit()        //Quit イベントが Finder からやってきたとき、
                                //あるいはアップルメニューの「終了」を選択したとき
{
        System.out.println("Quit Event Received");
        System.exit(0);   //アプリケーションの終了
}

public void handleAbout()     //アップルメニューの About を選択したとき
{
        System.out.println("Select About Menu");
}

/*TextViewer のウインドウが何も表示されていないときにもメニューを表示されるように
        したいそのような場合に必要なメニューに絞って表示するが、メニューを表示するには
        そもそも Frame でなければならない
        このクラスは Frame を拡張したが、そのウインドウは表示したくないので、
        コンストラクタで画面外に追いやった   */
private void setupMenuBar()
{
        /*メニュー作成部分のコメントは TextViewer.java を参照*/

        MenuItem newItem, openItem, closeItem, quitItem;

        MenuBar mb = new MenuBar();

        Menu fileMenu = new Menu("ファイル", true);
        newItem = new MenuItem("新規", new MenuShortcut('N'));
        fileMenu.add(newItem);
        openItem = new MenuItem("開く...", new MenuShortcut('O'));
        fileMenu.add(openItem);
        fileMenu.addSeparator();
```

```
        quitItem = new MenuItem("終了", new MenuShortcut('Q'));
        fileMenu.add(quitItem);
        mb.add(fileMenu);

        NewItemListener newListener = new NewItemListener();
        newItem.addActionListener(newListener);

        OpenItemListener openListener = new OpenItemListener();
        openItem.addActionListener(openListener);

        QuitItemListener quitListener = new QuitItemListener();
        quitItem.addActionListener(quitListener);

        setMenuBar(mb);
    }

    class NewItemListener implements ActionListener  {
        public void actionPerformed(ActionEvent ev)  {
            new TextViewer(null);
        }
    }

    class OpenItemListener implements ActionListener {
        public void actionPerformed(ActionEvent ev)  {
            String filename = null;
            FileDialog stdDlg =
                new FileDialog(Draggable.this, "開くファイルを指定してください");
            stdDlg.show();
            filename = stdDlg.getFile();
            if(filename != null)
                new TextViewer(new File(stdDlg.getDirectory(), filename));
        }
    }

    class QuitItemListener implements ActionListener  {
        public void actionPerformed(ActionEvent ev)  {
            System.exit(0);
        }
    }

}
```

TextViewer.java

```
import java.awt.*;
import java.awt.event.*;
import java.awt.datatransfer.*;
import java.io.*;
import java.util.*;

public class TextViewer extends Frame
```

```
    implements WindowListener, ComponentListener, ClipboardOwner
{
    File openFile;    //開いているファイル
    Dimension initilaWindowSize = new Dimension(500,300);        //ウインドウの初期サイズ
    TextArea viewText;    //ウインドウ内に配置する TextArea

    public TextViewer(File targetFile)
    {
        openFile = targetFile;                //引数のファイルをメンバ変数に保存

        /*イベントリスナの組み込み*/
        addWindowListener(this);            //WindowListenner を組み込む
        addComponentListener(this);        //ComponentListener を組み込む

        /*ウインドウの表示など、ウインドウの初期化作業*/
        show();                                //ウインドウを表示する
        setSize(initilaWindowSize);        //ウインドウのサイズを初期サイズにする
        if (openFile != null)    {            //ファイルが指定されているかをチェック
            setTitle(openFile.getName());      //ウインドウのタイトルはファイル名にする
        }
        setLayout(null);                        //レイアウト機能は利用しない

        /*テキストを表示するための TextArea の用意*/
        viewText = new TextArea();  //TextArea を新たに用意する
        viewText.setBounds(0,0,initilaWindowSize.width,initilaWindowSize.height);
                                    //TextArea をウインドウの大きさぴったりに合わせる
        add(viewText);    //TextArea をウインドウに追加する

        /*ファイルから読み込んだ内容を TextArea に設定する*/
        if (openFile != null)    {        //ファイルが指定されているかをチェック
            StringBuffer fText = new StringBuffer("");        //StringBuffer を用意する
            String aLine;                //ファイルから読み込んだ 1 行分を保持する変数を定義
            try    {
                LineNumberReader LNR =
                        new LineNumberReader(new FileReader(targetFile));
                            //引数に指定したファイルから行読み込みする  Reader  を生
成
                while(true) {        //とりあえず無限ループ
                    aLine = LNR.readLine();      //Reader から 1 行読み込む
                    if(aLine == null)          break;
                        //最後の行まで読み込むと null になるのでループを抜ける
                    fText.append(aLine + "¥n");
                        //読み込んだ 1 行分に改行を追加し、StringBuffer に追加する
                };
                LNR.close();      //Reader を閉じる
            }
            catch(Exception e)    {        //例外があったときの処理
                System.out.println(e.getMessage());
                                    //エラーメッセージをコンソールに表示
            }
            viewText.setText(fText.toString());
                                    //ファイルから読み込んだテキストを TextArea に設定する
        }
```

```
        setupMenuBar(); //メニューを構築する
    }

    private void setupMenuBar()
    {
        MenuItem newItem, openItem, saveItem, saveAsItem, closeItem, quitItem;
            //「ファイル」メニューの項目
        MenuItem undoItem, cutItem, copyItem, pasteItem; //「編集」メニューの項目
        MenuItem bgWhite, bgGray, bgYellow;
            //「編集」メニューの「背景色」のサブメニュー項目
        MenuItem aboutMRJ;          //「ヘルプ」メニューに追加する項目

        MenuBar mb = new MenuBar();    //メニューバーを用意する

        /*「ファイル」メニューの作成*/
        Menu fileMenu = new Menu("ファイル", true);        //「ファイル」メニューを作成する
        newItem = new MenuItem("新規", new MenuShortcut('N'));
                                        //「新規」項目を作成、ショートカットは N
        fileMenu.add(newItem);          //「ファイル」メニューに追加する
        openItem = new MenuItem("開く...", new MenuShortcut('O'));
                                        //「開く」項目を作成、ショートカットは O
        fileMenu.add(openItem);         //「ファイル」メニューに追加する
        closeItem = new MenuItem("閉じる", new MenuShortcut('W'));
                                        //「閉じる」項目を作成、ショートカットは W
        fileMenu.add(closeItem);        //「ファイル」メニューに追加する
        fileMenu.addSeparator();        //区切り線を追加する
        saveItem = new MenuItem("保存", new MenuShortcut('S'));
                                        //「保存」項目を作成、ショートカットは S
        saveItem.setEnabled(false); //「保存」を選択できないようにイネーブルでなくす
        fileMenu.add(saveItem);         //「ファイル」メニューに追加する
        saveAsItem = new MenuItem("名前を付けて保存...");
                                    //「名前を付けて保存」項目を作成、ショートカットはなし
        saveAsItem.setEnabled(false);
                                //「保存を付けて保存」を選択できないようにイネーブルでなくす
        fileMenu.add(saveAsItem);       //「ファイル」メニューに追加する
        fileMenu.addSeparator();        //区切り線を追加する
        quitItem = new MenuItem("終了", new MenuShortcut('Q'));
                                        //「終了」項目を作成、ショートカットは Q
        fileMenu.add(quitItem);         //「ファイル」メニューに追加する
        mb.add(fileMenu);               //「ファイル」メニューをメニューバーに追加する

        NewItemListener newListener = new NewItemListener();
                        //「新規」を選択したときにイベントを受け付けるクラスを新たに生成
        newItem.addActionListener(newListener);
                        //「新規」を選んだ時にイベントが発生するように設定

        OpenItemListener openListener = new OpenItemListener();
                        //「開く」を選択したときにイベントを受け付けるクラスを新たに生成
        openItem.addActionListener(openListener);
                        //「開く」を選んだ時にイベントが発生するように設定

        CloseItemListener closeListener = new CloseItemListener();
                        //「閉じる」を選択したときにイベントを受け付けるクラスを新たに生
成
```

```
closeItem.addActionListener(closeListener);
                    //「閉じる」を選んだ時にイベントが発生するように設定

SaveItemListener saveListener = new SaveItemListener();
                    //「保存」「保存を付けて保存」を選択したときに
                    //イベントを受け付けるクラスを新たに生成
saveItem.addActionListener(saveListener);
                    //「保存」を選んだ時にイベントが発生するように設定
saveAsItem.addActionListener(saveListener);
                    //「保存を付けて保存」を選んだ時にイベントが発生するように設定

QuitItemListener quitListener = new QuitItemListener();
                    //「終了」を選択したときにイベントを受け付けるクラスを新たに生成
quitItem.addActionListener(quitListener);
                    //「終了」を選んだ時にイベントが発生するように設定

/*「編集」メニューの作成*/
Menu editMenu = new Menu("編集", true);      //「編集」メニューを作成する
undoItem = new MenuItem("やり直し", new MenuShortcut('Z'));
                                        //「やり直し」項目を作成、ショートカットは Z
undoItem.setEnabled(false);
                        //「やり直し」を選択できないようにイネーブルでなくす
editMenu.add(undoItem);             //「編集」メニューに追加する
editMenu.addSeparator();            //区切り線を追加する
cutItem = new MenuItem("カット", new MenuShortcut('X'));
                                        //「カット」項目を作成、ショートカットは X
editMenu.add(cutItem);              //「編集」メニューに追加する
copyItem = new MenuItem("コピー", new MenuShortcut('C'));
                                        //「コピー」項目を作成、ショートカットは C
editMenu.add(copyItem);             //「編集」メニューに追加する
pasteItem = new MenuItem("ペースト", new MenuShortcut('V'));
                                        //「ペースト」項目を作成、ショートカットは V
editMenu.add(pasteItem);            //「編集」メニューに追加する
editMenu.addSeparator();            //区切り線を追加する

/*「編集」メニュー「背景色」でのサブメニューの作成*/
Menu bgColorMenu = new Menu("背景色");
                                        //サブメニューの元になる「背景色」項目を作成
bgWhite = new MenuItem("白");           //「白」項目を作成
bgColorMenu.add(bgWhite);               //「背景色」のサブメニューに追加する
bgGray = new MenuItem("グレイ");        //「グレイ」項目を作成
bgColorMenu.add(bgGray);                //「背景色」のサブメニューに追加する
bgYellow = new MenuItem("黄色");            //「黄色」項目を作成
bgColorMenu.add(bgYellow);              //「背景色」のサブメニューに追加する
editMenu.add(bgColorMenu);
                        //「背景色」のサブメニューを「編集」メニューに追加する

mb.add(editMenu);       //「編集」メニューをメニューバーに追加する
setMenuBar(mb);         //メニューバーを Frame に設定する

EditItemListener editListener = new EditItemListener();
        //「編集」メニューを選択した時の処理を行うクラスを新たに生成する
undoItem.addActionListener(editListener);
        //「やり直し」を選択したときにイベントが発生するようにしておく
```

```
        cutItem.addActionListener(editListener);
            //「カット」を選択したときにイベントが発生するようにしておく
        copyItem.addActionListener(editListener);
            //「コピー」を選択したときにイベントが発生するようにしておく
        pasteItem.addActionListener(editListener);
            //「ペースト」を選択したときにイベントが発生するようにしておく

        BackgroundColorItemListener bgListener = new BackgroundColorItemListener();
            //「背景色」のサブメニューを選択した時の処理を行うクラスを新たに生成する
        bgWhite.addActionListener(bgListener);
            //「白」を選択したときにイベントが発生するようにしておく
        bgGray.addActionListener(bgListener);
            //「グレイ」を選択したときにイベントが発生するようにしておく
        bgYellow.addActionListener(bgListener);
            //「黄色」を選択したときにイベントが発生するようにしておく

        /*「ヘルプ」メニューへの追加*/
        Menu helpMenu = new Menu("ヘルプ"); //「ヘルプ」メニューを作成する
        helpMenu.add(aboutMRJ = new MenuItem("MJR とは？"));
                                //「MJR とは？」という項目を追加する
        mb.add(helpMenu);    //「ヘルプ」メニューをメニューバーに追加する
        mb.setHelpMenu(helpMenu);       //ここで作成したメニューをヘルプメニューとする
            //これにより、システムが用意する「ヘルプ」メニューと一体化する
            //ただし、MRJ2.1EA3 ではなぜか余計な項目の「X」というのも追加される。
            //これはバグだと思われる

        aboutMRJ.addActionListener(
                //ヘルプメニューに追加した項目を選択したときの処理を直接記述している
            new ActionListener()          {
                public void actionPerformed(ActionEvent ev)  {
                    System.out.println("Help");
                                            //処理は単にコンソールに文字を出力するだけ
                }     }     );
}

/*「ファイル」メニューの「新規」を選択したときの処理*/
class NewItemListener implements ActionListener  {
    public void actionPerformed(ActionEvent ev)  {
        new TextViewer(null);  //TextViewer のウインドウを新たに作成する
    }
}

/*「ファイル」メニューの「開く」を選択したときの処理*/
class OpenItemListener implements ActionListener {
    public void actionPerformed(ActionEvent ev)  {
        FileDialog stdDlg =
            new FileDialog(TextViewer.this, "開くファイルを指定してください");
                //ファイル選択ダイアログボックスを生成する
        stdDlg.show();    //ダイアログボックスを表示する
        String filename = stdDlg.getFile(); //選択したファイル名を取得
        if(filename != null)       //ファイルを選択したのなら
            new TextViewer(new File(stdDlg.getDirectory(), filename));
                //ディレクトリと合わせて File を生成し、
                //それを開く TextViewer を新たに作成
```

```
            }
/*これにより、新たなウインドウを開くことができるはずだが、制約がある。FileDialog で取得したファイル名やディレクトリは、URL エンコードされた文字列になっている。MRJ2.1 では、ファイル名は URL エンコードしないで UNICODE 文字列そのままで指定するので、結果的に FileDialog で得られたファイル情報からはそのままファイルを開くことはできない。したがって、現状では、URL エンコードしても変化しないファイル名のものしか開くことはできない。
もちろん、URL デコードして値を引き渡せばいいのだが、たぶん、FileDialog で URL エンコードした結果が得られるのは、バグか機能の組み込みが遅れているためだと思われる。待っていれば、ファイル名などを通常の文字列として得られるようになると思われるので、ここでは機能は組み込まなかった。*/
    }

    /*「ファイル」メニューの「閉じる」を選択したときの処理*/
    class CloseItemListener implements ActionListener {
        public void actionPerformed(ActionEvent ev)  {
            dispose();          //Frame を閉じる
        }
    }

    /*「ファイル」メニューの「保存」「名前を付けて保存」を選択したときの処理*/
    class SaveItemListener implements ActionListener  {
        public void actionPerformed(ActionEvent ev)  {
            /*      未作成       */
        }
    }

    /*「ファイル」メニューの「終了」を選択したときの処理*/
    class QuitItemListener implements ActionListener   {
        public void actionPerformed(ActionEvent ev)  {
            System.exit(0);
        }
    }

    /*「編集」メニューを選択したときの処理*/
    class EditItemListener implements ActionListener   {
        public void actionPerformed(ActionEvent ev)  {
            String itemLabel = ((MenuItem)ev.getSource()).getLabel();
                    //選択したメニューの項目名を取り出す
            Clipboard sysClip = getToolkit().getSystemClipboard();
                    //システムのクリップボードを参照する
            if(itemLabel.compareTo("やり直し") == 0)            //選択したのが「やり直し」なら
                /*      難しい...    */;
            else if(itemLabel.compareTo("カット") == 0)  {      //選択したのが「カット」なら
                StringSelection selStr = new StringSelection(viewText.getSelectedText());
                    //TextArea の選択範囲の文字列から、クリップボードにセットできる
                    //形式のオブジェクトを生成する
                sysClip.setContents(selStr, TextViewer.this);
                                                                //クリップボードにセットした
                int selStart = viewText.getSelectionStart();    //選択範囲の最初の位置
                int selEnd = viewText.getSelectionEnd();        //選択範囲の最後の位置
                viewText.replaceRange("", selStart, selEnd);  //選択範囲を削除しておく
            }
            else if(itemLabel.compareTo("コピー") == 0)  {
```

```
                    StringSelection selStr = new StringSelection(viewText.getSelectedText());
                        //TextArea の選択範囲の文字列から、クリップボードにセットできる
                        //形式のオブジェクトを生成する
                    sysClip.setContents(selStr, TextViewer.this);
                                                            //クリップボードにセットした
                }
                else if(itemLabel.compareTo("ペースト") == 0){      //選択したのが「ペースト」な
ら
                    Transferable contents = sysClip.getContents(this);
                                                //クリップボードからデータを取り出す
                    if((contents != null) &&
                            (contents.isDataFlavorSupported(DataFlavor.stringFlavor)))
    {
                                        //データが存在し、かつそのデータがテキストならば
                        int selStart = viewText.getSelectionStart();      //選択範囲の最初の位置
                        int selEnd = viewText.getSelectionEnd();        //選択範囲の最後の位置
                        try    {
                        viewText.replaceRange(
                            (String)contents.getTransferData(DataFlavor.stringFlavor),
                            selStart, selEnd);
                                //クリップボード内のテキストデータを、
                                //TextArea の選択範囲に置き換える
                        }
                        catch (Exception e)     {System.out.println(e.getMessage());    }
                    }
                }
            }
    }

    /*「編集」メニューの「背景色」から項目をを選択したときの処理*/
    class BackgroundColorItemListener implements ActionListener         {
        public void actionPerformed(ActionEvent ev)  {
            String itemLabel = ((MenuItem)ev.getSource()).getLabel();
                                                //選択したメニューの項目名を取り出す
            if(itemLabel.compareTo("白") == 0)       {              //選択したのが「白」なら
                viewText.setBackground(Color.white);}  //TextArea の背景を白にする
            else if(itemLabel.compareTo("グレイ") == 0){  //選択したのが「グレイ」なら
                viewText.setBackground(Color.gray);}    //TextArea の背景をグレイにする
            else if(itemLabel.compareTo("黄色") == 0)      {       //選択したのが「黄色」なら
                viewText.setBackground(Color.yellow);} //TextArea の背景を黄色にする
            viewText.repaint();
        }
    }

    /*ウインドウのサイズを変更した時に呼び出される*/
    public void componentResized(ComponentEvent e)
    {
        System.out.println("Window Resized");
        viewText.setSize(this.getSize());
                        //TextArea の大きさをウインドウの変更後のサイズに合わせる
    }

    public void componentMoved(ComponentEvent e){}
    public void componentShown(ComponentEvent e){}
```

```
    public void componentHidden(ComponentEvent e){}

    /*ウインドウのクローズボックスをクリックしたとき*/
    public void windowClosing(WindowEvent e)
    {
        dispose();  //ウインドウを閉じる。
                    //これがないとクローズボックスをクリックしてもウインドウは閉じない
    }
    public void windowOpened(WindowEvent e){}
    public void windowClosed(WindowEvent e){}
    public void windowIconified(WindowEvent e){}
    public void windowDeiconified(WindowEvent e){}
    public void windowActivated(WindowEvent e){}
    public void windowDeactivated(WindowEvent e){}

    /*印刷のときに呼び出されるメソッド。単に印刷すれば、画面通りに印刷されるが、
        ここでの定義により、TextArea 内のテキストを印刷することができる*/
    public void printAll(Graphics g)
    {
        String targetLine;
        int currentBase = 0;
        int LinePitch = 16;
        StringTokenizer eachLines = new StringTokenizer(viewText.getText(), "¥n", false);
        while(eachLines.hasMoreTokens())       {
            targetLine = eachLines.nextToken();
            currentBase += LinePitch;
            g.drawString(targetLine, 0, currentBase);
        }
    }

/*ClipboardOwner をインプリメントした時にはこのメソッドをオーバーライドしなければならない*/
    public void lostOwnership(Clipboard cb, Transferable tr)
    {
        System.out.println("Lost Clipboard Ownership");
    }
}
```

第6章　基本的なファイルの処理

● アプリケーションソフトでは欠かすことができないファイル処理ですが、基本的なファイルの処理については、Java のライブラリである java.io を利用して行えます。java.io でのファイル処理の概略について説明します。

● ただし、java.io に必要な処理が全て揃っているわけではありません。非表示ファイルかどうかの判断や、あるいはファイルタイプやクリエイター、Finder 情報といった Mac OS 独自の機能についての処理は java.io にはありません。

● MRJ Toolkit の機能や、JDirect として提供される Mac OS の Toolbox を呼び出す機能を利用して、java.io にない機能も実現してみます。

● MRJ2.1EA3 をベースに動作確認した結果で解説を行います。

6-1　java.io でのファイルの取り扱い

Java の標準ライブラリとして提供される java.io には、ファイルの入出力やファイル管理のためのクラスが用意されていて、基本的なファイルの入出力処理などができるようになっています。OS に依存しないような処理だけということになりますが、Java でアプリケーションを作る時には、java.io に用意された機能を使うことがまず大前提になります。java.io に用意された機能のうち、テキストファイルの入出力に必要なものに絞って概略を説明しましょう。

File クラス

ディスク装置などに存在する 1 つのファイルやフォルダを、Java のプログラム上で 1 つのオブジェクトとして扱うのが、java.io にある File クラスです。名前は File ですが、フォルダ（ディレクトリ）についても、File クラスで処理することができます。

既存のファイルを参照する File オブジェクトを用意したり、あるいはこれから作成するファイルの File オブジェクトを生成するということが実際に行われるでしょう。その場合、new を利用して File クラスのコンストラクタを呼び出し、新たにオブジェクトを生成します。通常は、File(String) 形式のコンストラクタを呼び出し、引数にはフルパス名の文字列を指定します。Mac OS ではボリューム参照番号やディレクトリ ID などを使いますが、Java の世界ではパスを文字列で記述することでファイルを指定します。他には、File(String, Stirng) 形式のコンストラクタがあり、フォルダとファイル名をそれぞれ文字列で指定します。また、File(File, String)形式のコンストラクタでは存在するフォルダを第 1 引数、ファイル名を第 2 引数に指定して、File オブジェクトを生成します。

♦ファイル名の指定の規則

Java の環境ではファイルの指定を文字列で行うのですが、その文字列の記述をどうするかが問題になります。MRJ2.1 が稼動する環境では、ファイル名の規則としては次のようになっています。

- **ファイル名、フォルダ名とも、基本的には Finder で見える通りの文字列を記述すればよい（以下の 2 文字は例外）**

- **ただし、ファイル名に含まれるスラッシュは「%2f」と URL エンコードによる記述を行う**
- **さらに、ファイル名に含まれるパーセントは「%25」と記述する必要がある**
- **フルパス名の冒頭はスラッシュ「/」で始まる**
- **ボリューム名、フォルダ名、ファイル名をスラッシュで区切る**

なお、MRJ2.0 までは、2 バイト文字についても、URL エンコードされたものを、文字列として指定する必要がありました。そのため、逆に特定のフォルダのファイル一覧を得たとしても、漢字や仮名の部分の URL エンコード文字列からシフト JIS のコードを取り出し、さらに UNICODE 文字列にする必要がありました。しかしながら、MRJ2.1 では上記のように、半角のスラッシュとパーセント以外は URL エンコードされませんので、ファイル名が比較的扱いやすくなっています。ただし、MRJ2.0 でのファイル名の扱いをプログラムに機能として組み込んだ場合には、MRJ2.1 向けに変更が必要になるかも知れません。

実際に存在するファイルやあるいはこれから作成するファイルを参照するための File オブジェクトを生成するプログラムは、たとえば次のようなになります。

```
import java.io.*;
    :
File targetFile
 = new File("/macos_system/書類/MJR/MJR Samples/ch06_Files/テキストファイル");
                //ファイルの絶対パスの指定方法はこの通り。スラッシュで始まる
                //*********実際に実行する環境のパスに変更してください*******
```

このプログラムでは、Mac OS が稼動する Macintosh の「macos_system」というボリュームにある、「書類」「MJR」「MJR Samples」「ch06_Files」とフォルダをたどって行った先にある「テキストファイル」という名前のファイルを参照します。このように、絶対的なファイルへのパスを文字列として記述するのが 1 つの代表的な方法です。最初のスラッシュを忘れないようにしてください。

なお、Mac OS では、ファイル名やフォルダ名は 31 バイトまでという制約があります。File オブジェクトを生成するときには特にそうした規則は考慮しません。仮に 31 バイトを超えるファイル名を指定して、その File オブジェクトに従ってファイルを作成した場合、一切のエラーや例外は発生せず、単にファイル名の最初から 31 バイト分の名前のファイルを作成します。30 バイト目に 2 バイトコードがある場合には、上位バイトだけがファイル名に残り、ゴミキャラクタがファイル名の末尾に付いてしまいます。実用アプリケーションでは 31 バイト以下に制限するという手続きをプログラムに組み込む必要があるでしょう。

♦アプリケーションの存在するフォルダ

起動したアプリケーションが存在するフォルダは、「$APPLICATION」という記述をパス名に含めることで参照することが可能です。たとえば、アプリケーションが存在するフォルダと同じフォルダにある「テキストファイル」というファイルを参照する File オブジェクトを生成するには次のようなプログラムを作成します。

```
import java.io.*;
    :
targetFile = new File("/$APPLICATION/テキストファイル");
                    //アプリケーションが存在するフォルダにある
                    //「テキストファイル」というファイルを参照する File オブジェクトを生成
```

アプリケーションが使うデータや設定などを、アプリケーションと同じフォルダのファイルやあるいはさらにそこにあるフォルダ内に置く場合、それらのファイルへの参照は比較的簡単に記述できるということです。

テキストファイルの読み書き

java.io にはたくさんのクラスが定義されていますが、それらは、汎用的なストリーミングの入出力を行うためのクラスであり、さまざまなバリエーションの機能を持ったクラスが発展形として用意されています。ここでは、一般的によく使われると思われるテキストファイル処理に絞り込んで、そこで使われるクラスとメソッドについて説明をしましょう。

♦テキストファイルからの読み込み

テキストファイルからの読み込みですが、ファイルからの読み込みでは、通常は FileReader というクラスを用意します。コンストラクタはいくつかありますが、FileReader(File) 形式のものを使うのがほとんどでしょう。つまり、フルパス名から File オブジェクトを生成しておき、その File オブジェクトから FileReader を生成します。

さらに、1 行単位で読み込みを行う機能を持った LineNumberReader というファイル読み込みのためのクラスを利用すると、テキストファイルではさらに便利です。FileReader では、ファイルから 1 文字あるいは指定した文字数を取り出すような read メソッドが利用できます。さらに LineNumberReader では、1 行分を文字列として読み込むという readLine メソッドが利用できるので、テキストファイルの扱いが非常にやりやすくなります。

LineNumberReader オブジェクトを生成することが、ファイルのオープンに相当します。そして、テキストファイルでは、readLine を使って行を読み込むことになりま

す。戻り値が読み込んだ 1 行文のテキストになります。最後の行まで読んだかどうかの判定は、readLine の戻り値が null かどうかで行います。

たとえば、以下のようなプログラムを利用すると、「テキストファイル」という名前のテキストファイルを読み込み、各行の文字列をコンソールに出力します。最後に close メソッドを使って、開いたファイルを閉じておきます。なお、readLine はファイル内の 1 行分を読み込みますが、改行コードまでは読み込みません。

```
import java.io.*;
     :
File targetFile
 = new File("/macos_system/書類/MJR/MJR Samples/ch06_Files/テキストファイル");
                  //ファイルの絶対パスの指定方法はこの通り。スラッシュで始まる
                  //*********実際に実行する環境のパスに変更してください*******
try  {
    LineNumberReader inFile = new LineNumberReader(new FileReader(targetFile));
                  //行読み込みを行えるように開く
    String aLine = inFile.readLine();    //1 行分のテキストを読む
    while(aLine != null)     {       //ファイルの最後まで読んだら null を戻す
          System.out.println(aLine);    //読んだテキストをコンソールに出力
          aLine = inFile.readLine();    //1 行分のテキストを読む
    }
    inFile.close();     //ファイルを閉じる
}
catch(Exception e)   {
    System.out.println("Exception:" + e.getMessage());
}
```

ファイルの中に 2 バイトコードがあれば、Mac OS の場合は基本的にはシフト JIS コードとして保存されていることになります。LineNumberReader で開いたファイルから readLine でテキストを読み込むと、ファイルのエンコーディングを考慮して変換を行い、戻り値は UNICODE、つまり Java のプログラムで普通に使える文字列として得られます。

表 6-1　LineNumberReader **クラスで利用できるメソッド**

戻り値	メソッド	機能
コンストラクタ	LineNumberReader(Reader)	行読み込みのリーダーを生成
	LineNumberReader(Reader, int)	行読み込みのリーダーを生成。第 2 引数はバッファサイズ
void	close()	閉じる
int	read()	1 文字分を読み取り、その文字コードを戻す。改行は Mac OS でも 10 が得られる。漢字なども含めて UNICODE のコード番号が得られる。ファイルの末

戻り値	メソッド	機能
		尾まで読むと-1 を戻す
int	read(char[], int, int)	ファイルから読み取ったデータを char 配列に代入する。第 2 引数で配列の最初の要素を指定し、第 3 引数で読み取る文字数を指定する。読み取った文字数を戻すが、-1 が戻された時には読み込みはしていない
String	readLine()	ファイルから 1 行分を読み取り戻す
long	skip(long)	引数に指定した文字数だけ読み飛ばす
void	setLineNumber(int)	現在の行番号を引数の数値で指定する
int	getLineNumber()	現在の行番号を得る
void	mark(int)	現在の読み取り位置にマークを設定する
void	reset()	いちばん近いマークを、現在の読み取り位置にする

♦テキストファイルへの書き込み

テキストファイルを書き込む場合には、ファイルの書き込みに用意されている FileWriter というクラスを生成すれば良いでしょう。FileReader と同様に、File クラスを引数にしたコンストラクタが用意されています。このとき、存在するファイルの File オブジェクトを引数に指定することも可能で、通常はファイルは上書きされます。また、ファイルのパスを示す文字列を引数に取ったコンストラクタも利用できます。FileWriter(String) 形式のコンストラクタもありますが、FileWriter(String, boolean) 形式のコンストラクタを使い、第 2 引数を true に指定すると、アペンドモードで開かれ、ファイルの末尾に追加するということができます。また、File やパスの文字列に存在しないファイルを指定してもかまいません。その場合、ファイルを新たに作成します。

FileWrite クラスの生成で、ファイルを開きます。その後、write メソッドを使ってデータをファイルに書き出します。write メソッドはいくつかのバリエーションがありますが、FileWriter の何段階か親クラスである Writer クラスでメソッドは定義されています。引数としては、write(String)、write(String, int, int)などいくつかバリエーションがありますが、テキストファイルを出力するとなると、String 型の引数を取るものを使うのが一般的でしょう。

以下のプログラムでは、「テキストファイル」というファイルからテキストを読み込み、1 行単位で「テキストファイルの出力」というファイルに書き出しを行っています。このとき、行数を変数でカウントして、出力するファイルのテキストには、行番号と行の内容を出力するようにします。

```
import java.io.*;
   :
```

```
targetFile = new File("/$APPLICATION/テキストファイル");
        //アプリケーションが存在するフォルダにある「テキストファイル」
        //というファイルを参照する File オブジェクトを生成
File outputFile = new File("/$APPLICATION/テキストファイルの出力");
        //アプリケーションが存在するフォルダにある「テキストファイルの出力」という
        //ファイルを参照する File オブジェクトを生成
String lineSeparator = System.getProperty("line.separator");
        //行の区切り文字列をシステムプロパティから取得
try {
    LineNumberReader inFile = new LineNumberReader(new FileReader(targetFile));
    FileWriter outFile = new FileWriter(outputFile);
                                    //テキスト出力のためにライターを開く
    int lCounter = 1;               //行数カウンタ
    String aLine = inFile.readLine();   //1 行分のテキストを読む
    while(aLine != null)    {
        System.out.println(aLine);  //コンソールに出力
        outFile.write(String.valueOf(lCounter));  //ファイルに現在の行番号を書き出す
        outFile.write(":");
        outFile.write(aLine);   //テキストファイルから読み込んだテキストをファイルに書き出す
        outFile.write(lineSeparator); //改行をテキストファイルに書き出す
        aLine = inFile.readLine();
        lCounter++;             //カウンタを進める
    }
    inFile.close();
    outFile.close();
}
catch(Exception e)  {
    System.out.println("Exception:" + e.getMessage());
}
```

なお、LineNumberReader の readLine メソッドでは 1 行文の文字列だけを取得します。そのため、FileWriter の 1 行分の出力を行う時、読み込んだテキストを出力するだけではなく、改行のコードも出力する必要があります。このプログラムでは、システムプロパティから改行コードを取得し、それを出力しています。

表 6-2　FileWriter **で使えるメソッドの要約**

戻り値	メソッド名	機能
コンストラクタ	FileWriter(String)	引数にファイルのパスを指定する
	FileWriter(String, boolean)	引数にファイルのパスを指定。第 2 引数を true にするとアペンドで開く
	FileWriter(File)	引数に File オブジェクトを指定する
	FileWriter(FileDecripter)	引数に FileDecripter オブジェクトを指定する
void	close()	閉じる
void	write(char[])	char 配列を書き出す
void	write(char[], int, int)	char 配列の一部を書き出す。第 2 引数で示す位置から、第 3 引数で示す数の要素までを書き出す
void	write(int)	引数に指定した文字コードの文字を出力する。たと

戻り値	メソッド名	機能
		えば write(100)は「d」という文字を出力する
void	write(String)	引数に指定した文字列を書き出す
void	write(String, int, int)	引数に指定した文字列の一部を出力。第 2 引数で示す位置から、第 3 引数で示す数の要素までを書き出す

ファイル属性のメソッド

File オブジェクトから、ファイルの属性を得ることができるように、File クラスには以下のようなメソッドが定義されています。

表 6-3　File クラスに定義されたファイルやフォルダ属性のメソッド

戻り値	メソッド	機能
long	length()	ファイルの長さを戻す。バイト数が戻される。文字数ではない
String	getName()	ファイル名、フォルダ名を戻す
String	getParent()	ファイルやフォルダを含むフォルダのパスを戻す
String	getPath()	ファイルやフォルダのパスを戻す
String	getAbsolutePath()	ファイルやフォルダの絶対パスを戻す
String	getCanonicalPath()	標準形式に文字変換した絶対パスを戻すようだが、MRJでは getAbsolutePath と同様の結果になる
boolean	isAbsolute()	絶対パスで指定されているかどうかを戻す
boolean	exists()	ファイルが存在するかどうかを戻す
boolean	canWrite()	書き込み可能かどうかを戻す
boolean	canRead()	読み込み可能かどうかを戻す
boolean	isFile()	ファイルなら true を戻す
boolean	isDirectory()	フォルダなら true を戻す
long	lastModified()	ファイルの更新日を示す数値を戻す。一般には比較のために利用し、日付や時刻を求めるためには利用しない

ファイル処理のメソッド

ファイルの管理についてのさまざまなメソッドは、File クラスで定義されています。ファイルの削除などを行うにはこれらのメソッドを使うことになります。なお、list メソッドの使い方については、次の節で具体例を示します。

表 6-4　File クラスに定義されたファイルやフォルダ管理のメソッド

戻り値	メソッド	機能

戻り値	メソッド	機能
boolean	delete()	File オブジェクトで示すファイルを削除する。削除できれば true を戻す
boolean	renameTo(File)	引数に指定した File オブジェクトのファイル名に変更する。変更できれば true を戻す
boolean	mkdir()	File オブジェクトのパスで指定したフォルダを作成する。作成できれば true を戻す
boolean	mkdirs()	File オブジェクトのパスで指定したフォルダを作るが、それを含む必要なフォルダも作成する。作成できれば true を戻す
String[]	list()	File オブジェクトのフォルダにあるファイルやフォルダの一覧を文字列の配列として戻す
String[]	list(FilenameFilter)	File オブジェクトのフォルダにあるファイルやフォルダの一覧を文字列の配列として戻す。引数に指定したクラスで、一覧に含める内容を制限できる

6-2　Mac OS のファイル処理

基本的なファイル処理を java.io で用意されたクラスで処理をするとしても、それだけでは必要な処理はまかなえません。そのための 1 つの手段として、Mac OS では、Toolbox の機能を Java から直接呼び出すということを行います。具体的には File Manager の機能を使うことになりますが、そのための方法を説明しましょう。

ファイル処理の必要性と MRJ の機能

java.io の処理は前の節で見た通りですが、たとえば、非表示ファイルかどうかという判断がないため、list メソッドでファイル一覧を取得した時、フォルダのカスタムアイコンを保持する「Icon 」という名前のファイルも一覧されてしまいます。また、Mac OS 環境独自の属性である、ひな形、エイリアス、ロックなども得ることができません。こうした処理は、JDirect として、Java から Mac OS のシステム機能である Toolbox を直接呼び出すことで実現できるように MRJ では用意されています。

また、ファイルタイプやクリエイターといった、Mac OS 独自のファイル属性も、java.io ではサポートされていません。これについては、MRJToolkit の機能の一部として提供されています。ファイルタイプやクリエイターについては、次の節でまとめて説明します。

いずれにしても、MRJ が独自に提供するクラスライブラリがあって、そこで定義されているクラスの使い方を知っていれば、こうした java.io にはないようなファイル処理も組み込むことができます。JDirect かどうか、MRJToolkit かどうかは、分類の問題だけで、プログラム上の違いはまったくありません。ただし、MRJ2.1EA3 の段階では、JDirect はドキュメント化されていないという違いはあります。

ただし、もちろん、こうした処理を使ったアプリケーションは、そのままでは Windows や UNIX での利用はできません。Java で Mac OS 専用アプリケーションを作るのなら気にする必要はないことですが、クロスプラットフォーム対応にするためには、各 OS ごとの処理に分岐するなどの配慮が必要になるところです。

FSSpec クラス

Toolbox の File Manager では、ファイルの存在を取り扱うために FSSpec 構造体が定義されていました。いわば、java.io の File クラスに相当するような存在が FSSpec 構造体なのです。FSSpec 構造体は、ボリューム ID 番号、ディレクトリ ID 番号、そしてファイル名の文字列で、特定のファイルを指定する方法になっています。

この FSSpec 構造体が、そのまま Java のプログラムで利用できるようになっています。パッケージ的には、com.apple.mrj.macos.toolbox.FSSpec となっていますが、具体的には「機能拡張」フォルダの MRJLibraries フォルダ内にある MRJClasses.zip にバイトコードが埋め込まれています。従って、

- **MRJ が稼動している**
- **プロジェクトに、MRJClasses.zip を登録する**
- **ソースの冒頭で com.apple.mrj.macos.toolbox.FSSpec をインポートする**

という条件が満たされれば、作成しているプログラムソース内で、FSSpec クラスを利用できるということになります。

まず、FSSpec クラスのオブジェクトを生成するには次のようなメソッドを使います。File やあるいは絶対パスから作成できるので、生成はプログラム上では 1 行で済むでしょう。なお、他にいくつかコンストラクタは定義されていますが、実用上はこれらのコンストラクタで済むものと思われます。

表 6-5　FSSpec **のコンストラクタ**

コンストラクタ	利用方法
FSSpec(File)	引数の File を参照する FSSpec を生成
FSSpec(String)	引数に指定したパスのファイルを参照する FSSpec を生成
FSSpec(short, int, String)	ボリューム参照番号、ディレクトリ ID、ファイル名から FSSpec を生成する。FileManager の FSMakeFSSpec と同様なものと考えて良い

FSSpec クラスは、FileManager の FSSpec 構造体と同様、ファイルあるいはフォルダの参照に利用できます。以下のような基本的な処理メソッドが用意されています。FSSpec 構造体と同様に、ボリューム ID、ディレクトリ ID、ファイル名が FSSpec クラスのプロパティとして存在すると考えて良いようで、それらの取得や設定メソッドがあります。また、親フォルダの FSSpec を取得するメソッドもあります。

表 6-6　FSSpec クラスの基本的な処理メソッド

戻り値	メソッド	機能
File	toFile()	FSSpec からそのファイルの File オブジェクトを取得する
FSSpec	makeParent()	ファイルやフォルダが存在するフォルダ、つまり親フォルダの FSSpec を取得する
short	getVRefNum()	ボリューム参照番号を得る
void	setVRefNum(short)	引数の値をボリューム参照番号として設定する
ing	getParID()	ディレクトリ ID 番号を得る
void	setParID(int)	引数の値をディレクトリ ID 番号として設定する
String	getName()	ファイル名を得る
void	setName(String)	引数の文字列をファイル名として指定する

MRJ の JDirect では、さらに、com.apple.mrj.macos.generated.FSSepcStruct というクラスが定義されており、このクラスを引数に指定するような File Manager のファイル処理関数がメソッドとして提供されています。しかしながら、ファイルの読み書きは、java.io パッケージで用意されており、基本的にはよほど特殊なことをしない限りはそれらのメソッドまでを使うことはないと思われます。

ただし、ファイルのロックを行ったり、ロックを解除するような機能が、なぜか FSSpec クラスには用意されていません。必要なら、FSSpecStruct を使うことになるのですが、これは MRJ が発展段階で、機能の組み込みが間に合っていないためだと考えられます。この 6-2 節の最後に、File Manager 関数を直接利用する方法を説明します。

Finder 情報の取得

FSSpec を必要とする場面は、通常はファイルの Finder 情報を取得したい場合でしょう。Finder 情報は、File Manager レベルでは、FInfo という構造体で得られます。MRJ の JDirect 環境では、Finder 情報を管理するために FInfoStruct クラスが定義されています。パッケージ名を含むクラス名は、com.apple.mrj.macos.generated.FInfoStruct となります。FInfoStruct にもコンストラクタなどは定義されていますが、事実上、FSSpec クラスに定義された以下のメソッドを使って取得し、その値を変更して設定を行うということになると思われるので、コンストラクタを使う場面はないと思われます。単に FInfoStruct を作るには、引数のないコンストラクタを利用します。

表 6-7　FSSpec のクラスで、Finder 情報に関連するもの

戻り値	メソッド	機能
FInfoStruct	getFInfo()	FSSpec が参照するファイルの Finder 情報を取得して戻す
void	setFInfo(FInfoStruct)	引数に指定したオブジェクトの内容を、FSSpec が参照するファイルの Finder 情報として設定する

なお、Finder 情報は、ファイルに対するもので、フォルダに対しては取得できません。FSSpec がフォルダを参照している場合に getFInfo メソッドを適用すると、例外が発生してしまうので、注意が必要です。また、化け文字を含むファイル名の場合には、list メソッドでファイル名は取得できても、それを利用して作成した FSSpec に getFInfo を適用した段階で例外が発生します。2 バイト文字を含むファイル名で 31 バイト以上のものを指定して、2 バイトコードが泣き別れて化け文字が入ってしまったような状況では例外が発生する場合があり、注意が必要です。

♦Finder 情報にあるデータの扱い

こうして取得した FInfoStruct クラスは、以下のようなメソッドが利用できます。ファイルタイプやクリエイターもありますが、わざわざ Finder 情報を取得しなくても、FSSpec や File などから直接取得したり設定ができます。具体的には次の節を見てください。この中で実際によく利用するのは Finder フラグになると思われます。

表 6-8　FInfoStruct クラスのメソッド

戻り値	メソッド	機能
short	getFdFlags()	Finder フラグを取得する
void	setFdFlags(short)	引数の値を Finder フラグとして設定する
PointStruct	getFdLocation()	Finder 上での位置を取得する
void	setFdLocation(PointStruct)	Finder 上での位置を設定する
short	getFdFldr()	fdFldr の取得（この属性は Finder が使う）
void	setFdFldr(short fdFldr)	fdFldr の設定
int	getFdType()	ファイルタイプを取得する
void	setFdType(int)	ファイルタイプを設定する
int	getFdCreator()	クリエイターを取得する
void	setFdCreator(int)	クリエイターを設定する

たとえば、あるファイルの FSSpec クラスを生成し、それを利用して Finder 情報を取得するプログラムは次のようになります。FSSpec の情報や Finder フラグの値は、

コンソールに出力しています。

```
import com.apple.mrj.macos.toolbox.FSSpec;
import com.apple.mrj.macos.generated.FInfoStruct;

    :

targetFile = new File("/macos_system/書類/MJR/MJR Samples/ch06_Files/テキストファイル");
                    //存在するファイルのフルパスを指定して、File オブジェクトを作成
                    //*********実際に実行する環境のパスに変更してください*******

FSSpec targetSpec = new FSSpec(targetFile);     //File から FSSpec を作成した
    System.out.println(String.valueOf(targetSpec.getVRefNum()));
    System.out.println(String.valueOf(targetSpec.getParID()));
    System.out.println(targetSpec.getName());     //FSSpec の各メンバを見てみる

FInfoStruct finderInfos = targetSpec.getFInfo();     //Finder 情報を得た
    System.out.println(String.valueOf(finderInfos.getFdFlags()));
                                                    //Finder フラグをコンソールに出力
```

♦Finder フラグのビットを検出する

Finder フラグは、short つまり 16 ビットの整数値として利用できるようになっていますが実際に知りたいのは、各ビットの値です。特定のビットが 0 か 1 かを判断するための定数が Toolbox では定義されていましたが、同様に MRJ 環境でも定数が利用できます。そのために、com.apple.mrj.macos.generated.FinderConstants というクラスが定義されており、そのクラスの Static なクラス変数として、ビット判断用の定数が用意されています。値はいずれも int 型です。

表 6-9　Finder **フラグの判定用の** FinderConstants **クラス変数**

クラス変数	定数値	判定内容
kIsOnDesk	0x0001	未使用
kColor	0x000E	この定数でマスクした 3 ビットを使って、ラベルの番号を指定する
kIsShared	0x0040	マルチユーザーで利用できるアプリケーションかどうか
kHasNoINITs	0x0080	（不明）
kHasBeenInited	0x0100	BNDL 情報を Finder が読み取ったかどうか
kHasCustomIcon	0x0400	カスタムアイコンが設定されているかどうか
kIsStationery	0x0800	ひな形かどうか
kNameLocked	0x1000	ファイル名やフォルダ名の変更ができなくなっているかどうか
kHasBundle	0x2000	BNDL リソースがあるかどうか
kIsInvisible	0x4000	非表示かどうか
kIsAlias	0x8000	エイリアスかどうか

たとえば、java.io.File にある list メソッドで、表示されているファイルだけの一覧を取得したいとします。そのために、list メソッドの引数に指定するフィルタクラスを定義しますが、そこで Finder フラグを利用して、非表示になっていないかを判断しています。Finder 情報を取得し、FinderConstants.kIsInvisible と AND 演算を行って、非表示ビットが立っているかを判定しています。getFInfo はファイルにしか適用できないので、先立ってフォルダかどうかの判定をしていますが、ここは java.io.File に用意されたメソッドを利用しています。たとえば、以下のようなプログラムになります。

```
import com.apple.mrj.macos.toolbox.FSSpec;
import com.apple.mrj.macos.generated.FInfoStruct;
import com.apple.mrj.macos.generated.FinderConstants;

    :

File appDir = new File("/$APPLICATION");  //アプリケーションが存在するフォルダを参照
String itemList[] = appDir.list(new VisibleFilesOnly());
                                          //表示されているファイル名の一覧を取得する
for(int i=0 ; i<itemList.length; i++)     {     //一覧を順番にコンソールに出力する
    System.out.println(String.valueOf(i) +":"+itemList[i]);
}

    :

class VisibleFilesOnly implements FilenameFilter        //list メソッドで使うフィルタクラス
                          //表示されているファイルだけを取り出し、
                          //非表示ファイルやフォルダは含まれないようにする
{
    public boolean accept(File dir, String name)
                          //このメソッドが 1 つの項目を取得する度に呼び出される
    {
        File targetFile = new File(dir, name);          //対象ファイルの File オブジェクトを作成
        if (targetFile.isDirectory())   //それがフォルダなら
            return false;               //リストに含めない
        else                            //ファイルなら
        {
        FSSpec targetSpec = new FSSpec(targetFile);     //FSSpec オブジェクトを構築する
        FInfoStruct finderInfos = targetSpec.getFInfo();    //ファイルの Finder 情報を取得する
        if ((finderInfos.getFdFlags() & FinderConstants.kIsInvisible) != 0)
                                                        //非表示のフラグをチェック
            return false;       //フラグがセットされていればリストに含めない
        else
            return true;
        }
    }
}
```

File Manager 関数の直接的な利用

ファイルをロックしたりあるいは解除するメソッドが FSSpec クラスに用意されていませんので、そのための File Manager の関数を直接呼び出す必要があります。また、これに限らず、File Manager の関数を直接呼び出したいと考えるときもあるでしょう。JDirect では、File Manager の関数をクラスメソッドに持つクラス com.apple.mrj.macos.generated.FileFunctions が定義されています。メソッド名は File Manager の関数名と同一になっており、たとえば FSMakeFSSpec 関数は、FileFunctions.FSMakeFSSpec(…) と Java のプログラム中で記述すれば利用できます。

これら File Manager 関数のメソッドの引数は、基本的に関数と同様ですが、いろいろと注意が必要になるようです。

FileFunctions クラスのメソッドでは、ファイルを参照するためのクラスとして、FSSpec クラスではなく、com.apple.mrj.macos.generated.FSSpecStruct クラスを使います。FSSpecStruct は、FSSpec オブジェクトが生成されていれば作ることができますが、ここではファイル名を Pascal の文字列となる byte 配列で指定をしなければなりません。具体例はサンプルを見てください。このように、Java の世界のものを Toolbox のものに独自に変換して使わないといけないようで、基本的には com.apple.mrj.macos.generated パッケージのものは直接プログラマが使うことは想定していないと思われます。

以下の例は、ファイルのロックを行うプログラムです。ファイル名の文字列は FSSpec の getName メソッドで得られますが、それは UNICODE 文字列です。getBytes で Mac OS の場合には、シフト JIS で文字列を記述した結果の byte 配列が得られます。ただし、そのままでは、要素の最初から文字列データが始まる形式なので、冒頭にバイト数があってその後に文字列データが続く Pascal 文字列に変換してやらないといけません。そうすることで、FSMakeFSSpec 関数に指定できるファイル名の配列が作成できるという具合です。

```
import com.apple.mrj.macos.toolbox.FSSpec;
import com.apple.mrj.macos.generated.FSSpecStruct;
import com.apple.mrj.macos.generated.FileFunctions;

    :

targetFile = new File("/macos_system/書類/MJR/MJR Samples/ch06_Files/テキストファイル");
                //存在するファイルのフルパスを指定して、File オブジェクトを作成
                //*********実際に実行する環境のパスに変更してください*******
```

```
FSSpec targetSpec = new FSSpec(targetFile);     //File から FSSpec を作成した

short s;   //Toolbox 関数の戻り値を代入する変数。OSErr 値が戻される

                    //以下、ファイル名の文字列を pascal 文字列とした byte 配列を得る
byte[] fName = targetSpec.getName().getBytes(); //ファイル名の byte 配列を得る
byte[] fNameP = new byte[fName.length + 1];             //この配列に pascal 文字列を構成する
fNameP[0] = (byte)fName.length;                         //最初の要素は文字のバイト数
for( int i = 0 ;   i < fName.length ;   i++)            //2 つ目以降の要素に文字自体を入れる
    fNameP[i+1] = fName[i];

FSSpecStruct fs = new FSSpecStruct();       //新しく FSSpecStruct を用意
s = FileFunctions.FSMakeFSSpec(targetSpec.getVRefNum(), targetSpec.getParID(), fNameP, fs);
                    //FSSpec が参照するファイルの FSSpecStruct を作成
    System.out.println(String.valueOf(s));     //戻り値をコンソールへ
s = FileFunctions.FSpSetFLock(fs);              //ファイルにロックをかける
    System.out.println(String.valueOf(s));     //戻り値をコンソールへ
```

FileFunctions に存在するクラスメソッドについてのリファレンスも必要になるかもしれませんが、Macintosh Java Report では作成しないことにします。利用方法を検討した限りでは、本来は FSSpec クラスでこうした機能を利用した「ロックをかける」などのメソッドが用意されるものと思われるので、FileFunctios は基本的には内部利用のためのクラスだと思われるからです。

いずれにしてもどうしても使いたい Toolbox 関数がある場合だけに利用方法は限定されると思います。その場合は、このサンプルを参考にして、FileFunctions に定義されたメソッドと、File Manager の使い方のドキュメントをなどを見ながらプログラムを作成してください。MJRClasses.zip に定義されたクラスを参照するには、Visual Cafe のクラスブラウザを使うか、Code Warrior に付属する Class Wrangler を利用します。

6-3 ファイルタイプやクリエイターの設定

ファイルタイプやクリエイターの設定のための機能は MRJ でいろいろな方法が提供されています。それらの方法をここでまとめておきます。Java で作ったアプリケーションで文書ファイルを保存したとき、必ずファイルタイプやクリエイターの設定が必要になります。そうしないと、文書をダブルクリックしてアプリケーションを起動することや、アイコンの表示がうまくなされないことになります。

ファイルタイプを取り扱うクラス

まず、MRJToolkit に用意されたファイルタイプやクリエイターを扱う機能を紹介します。こちらの方が、ファイルタイプやクリエイターのための 4 文字キャラクタを扱うクラスが定義されているので、プログラム自体は作りやすい面もあるかもしれません。

ファイルタイプやクリエイターを扱うために、com.apple.mrj.MRJOSType というクラスが MRJ 環境で利用できるようになっています。このクラスを生成するコンストラクタには次の表のような 3 種類のパターンがあります。

表 6-10 MRJOSType のコンストラクタ

コンストラクタ	生成規則
MRJOSType(String)	引数には 4 文字のキャラクタを指定する
MRJOSType(int)	引数には整数を指定する。"TEXT" ならそのコードを並べた 0x54455854（整数値としては 1413830740）を引数に指定する
MRJOSType(byte[])	引数には byte 型の配列を指定する

文字列や byte 配列だけでなく、整数から生成する方法もあります。int でファイルタイプやクリエイターを指定する方法は表の中に記述した例を参照してください。MRJOSType には以下のようなメソッドも定義されています。

表 6-11 MRJOSType に用意されたメソッド

戻り値	メソッド	機能
boolean	equals(MRJOSType)	ファイルタイプやクリエイターの値を比較して同一なら true を戻す

戻り値	メソッド	機能
int	toInt()	ファイルタイプやクリエイターを int の形式で表現したデータを得る
String	toString()	ファイルタイプやクリエイターを文字列で表現したデータを得る

ファイルタイプやクリエイターの設定と取得

MRJ 環境では、com.apple.mrj.MRJFileUtils というクラスのクラスメソッドとして、ファイルタイプやクリエイターの設定や取得を行うものが用意されています。ファイルタイプやクリエイターは MRJOSType クラスのオブジェクトとして扱います。

表 6-12　ファイルタイプやクリエイター関連のメソッド

戻り値	メソッド	機能
MRJOSType	MRJFileUtils.getFileType (File)	引数に指定したファイルのファイルタイプを取得して戻す【static】
void	MRJFileUtils.setFileType (File, MRJOSType)	引数に指定したファイルのファイルタイプを、第 2 引数に指定したものに変更する【static】
MRJOSType	MRJFileUtils.getFileCreator(File)	引数に指定したファイルのクリエイターを取得して戻す【static】
void	MRJFileUtils.setFileCreator(File, MRJOSType)	引数に指定したファイルのクリエイターを、第 2 引数に指定したものに変更する【static】
void	MRJFileUtils.setFileTypeAndCreator (File, MRJOSType, MRJOSType)	引数に指定したファイルのファイルタイプとクリエイターを、それぞれ第 2、第 3 引数に指定したものに変更する【static】

いずれのメソッドも、IOException の例外をハンドリングしなければなりません。

java.io パッケージの FileWriter で新しくファイルを作った場合、特に指定をしていなければ、ファイルタイプは ????、クリエイターは作成したアプリケーションのクリエイターと同一になります。

たとえば、File オブジェクトを生成して、そのファイルに設定されたファイルタイプやクリエイターを取得するには次のようなプログラムを作成します。取得した結果はコンソールに出力するようになっています。ここでは既定のファイルタイプとクリエイターがどうなるかを調べることになるでしょう。

```
import; com.apple.mrj.MRJOSType;
```

```
import com.apple.mrj.MRJFileUtils;

    :

File outputFile = new File("/$APPLICATION/テキストファイルの出力");
                //アプリケーションが存在するフォルダにある「テキストファイルの出力」という
                //ファイルを参照する File オブジェクトを生成

    /*    この間に「テキストファイルの出力」というファイルを作成して書き出す      */

try  {
    MRJOSType defaultType = MRJFileUtils.getFileType(outputFile);
                //作成したファイルのファイルタイプを得る
    System.out.println(outputFile.getName() + "のファイルタイプ：  " + defaultType.toString());
                //コンソールに出力
    MRJOSType defaultCreator = MRJFileUtils.getFileCreator(outputFile);
                //作成したファイルのクリエイターを得る
    System.out.println(outputFile.getName() + "のクリエイター：  " + defaultCreator.toString());
                //コンソールに出力
}
catch(Exception e)   {
    System.out.println("Exception:" + e.getMessage());
}
```

また、File オブジェクトを生成して、そのファイルのファイルタイプとクリエイターを変更するには、次のようなプログラムになります。ファイルタイプとクリエイターをそれぞれ設定するメソッドがありますが、ここではまとめて変更できるメソッドを使ってみました。

```
import; com.apple.mrj.MRJOSType;
import com.apple.mrj.MRJFileUtils;

    :

File outputFile = new File("/$APPLICATION/テキストファイルの出力");
                //アプリケーションが存在するフォルダにある「テキストファイルの出力」という
                //ファイルを参照する File オブジェクトを生成

    /*    この間に「テキストファイルの出力」というファイルを作成して書き出す      */

MRJOSType theType = new MRJOSType("TEXT");
                                        //付けるファイルタイプはテキストファイル
MRJOSType theCreator = new MRJOSType("ttxt");     //付けるクリエイターは SimpleText のもの
try  {
    MRJFileUtils.setFileTypeAndCreator(outputFile, theType, theCreator);
                //ファイルタイプとクリエイターをファイルに一気に設定する
}
catch(Exception e)   {
    System.out.println("Exception:" + e.getMessage());
}
```

既定値のファイルタイプやクリエイター

ファイルタイプやクリエイターを作成したファイルに対していちいち設定するのも 1 つの方法ですが、既定のファイルタイプやクリエイターを、以下の表のメソッドを使って指定することもできます。いずれも、com.apple.mrj.MRJFileUtils クラスに定義されたクラスメソッドです。

表 6-13　既定値のファイルタイプやクリエイターの設定

戻り値	メソッド	機能
void	MRJFileUtils.setDefaultFileType (MRJOSType)	既定のファイルタイプを引数に指定したものに設定する【static】
void	MRJFileUtils.setDefaultCreator (MRJOSType)	既定のクリエイターを引数に指定したものに設定する【static】

いずれのメソッドも、IOException の例外をハンドリングしなければなりません。

たとえば、以下のようなプログラムにより、「既定値を変えて作ったファイル」という名前のファイルが作成されますが、そのファイルタイプとクリエイターは、setDefaultFileType および setDefaultFileCreator メソッドで指定したものになります。

```
import com.apple.mrj.MRJFileUtils;

    :

try {
    MRJFileUtils.setDefaultFileType(new MRJOSType("TEXT"));
            //既定のファイルタイプとしてテキストファイル
    MRJFileUtils.setDefaultFileCreator(new MRJOSType("JEDT"));
            //既定のクリエイターとして Jedit のクリエイターを設定
    FileWriter outFile = new FileWriter("/$APPLICATION/既定値を変えて作ったファイル");
            //ファイルを新規に作成すると、既定値として指定したファイルタイプとクリエイターになる
    outFile.close();
}
catch(Exception e)  {
    System.out.println("Exception:" + e.getMessage());
}
```

既定のファイルタイプやクリエイターについては、アプリケーションが起動している間は有効になるようです。アプリケーションを終了すると、既定値の設定は無効になります。

既定のファイルタイプやクリエイターを設定したからと言って、FileWriter でファイルを開いたファイルのファイルタイプやクリエイターが必ず既定のものになるわけ

ではありません。既定のファイルタイプやクリエイターは、ファイルが存在していなくて、新しくファイルを作成したときに適用されます。たとえば、既存のファイルに対して FileWriter で書き込みを行った場合、そのファイルのファイルタイプやクリエイターはもとからファイルに設定されているものになります。

FSSpec を利用した処理

FSSpec クラスにもファイルタイプやクリエイターの取得や設定のためのメソッドが用意されています。文字列あるいは int でファイルタイプやクリエイターのデータを扱いますが、int の場合の値については、MRJOSType のコンストラクタでの解説を参照してください。以下のようにメソッドが用意されていますが、FSSpec オブジェクトに適用するメソッドだけでなく、クラスメソッドも用意されていて、ファイルタイプやクリエイターを変更するファイルを引数で指定する方法も取ることができます。別のファイルのファイルタイプやクリエイターをコピーするメソッドも利用できます。

表 6-14　FSSpec クラスで利用できるファイルタイプとクリエイターの処理

戻り値	メソッド	機能
int	getCreator()	クリエイターを取得する
void	setCreator(int)	引数に指定した値をクリエイターとして設定する
int	getType()	ファイルタイプを取得する
void	setType(int)	引数に指定した値をファイルタイプとして設定する
void	FSSpec.setType(File, int)	引数に指定したファイルのクリエイターを、第 2 引数に指定した値にする【static】
void	FSSpec.setType(File, String)	引数に指定したファイルのクリエイターを、第 2 引数に指定した値にする【static】
void	setTypeAndCreator(int, int)	ファイルタイプとクリエイターを int 型で設定する
void	setTypeAndCreator(int, boolean, int, boolean)	ファイルタイプとクリエイターを設定する（boolean の引数の意味は不明）
void	setTypeAndCreator(String, String)	ファイルタイプとクリエイターを文字列で設定する
void	FSSpec.setTypeAndCreator(File, int, int)	引数に指定したファイルのファイルタイプとクリエイターを int 型で設定する【static】

戻り値	メソッド	機能
void	FSSpec.setTypeAndCreator(File, String, String)	引数に指定したファイルのファイルタイプとクリエイターを文字列で設定する【static】
void	syncTypeAndCreatorWith(FSSpec)	適用したオブジェクトのファイルタイプとクリエイターを、引数に指定したファイルと同じものに変更する
void	syncTypeAndCreatorWith(File)	適用したオブジェクトのファイルタイプとクリエイターを、引数に指定したファイルと同じものに変更する
void	FSSpec.syncTypeAndCreator(File f1, File f2)	f2 のファイルタイプとクリエイターを f1 と同じものにする【static】

6-4　サンプルプログラム

この章で説明したことを検証するためのサンプルプログラムを紹介しておきます。アプリケーションとして実行する FileTest **というクラスを生成するソースファイルが** 1 **つだけあります。いろいろなテストをまとめて行い、結果はコンソールに表示するという地味なものですが、各機能の検証をこれで行いました。**

FileTest.java

サンプルプログラムは 1 つのソースにファイルにすべてがまとまっています。MRJが利用できるように、必要ならクラスファイルをプロジェクトに登録しておきます。また、プロジェクトはアプリケーションが作成されるようにしておきます。

配付するファイルでは、読み込むテキストファイルなどをあらかじめ用意しておきますが、フルパス名を指定している箇所では、実際にプログラムを稼動させる環境に合わせて、パス名の記述を変更しておく必要があります。

```
import java.io.*;
import com.apple.mrj.MRJOSType;
import com.apple.mrj.MRJFileUtils;
import com.apple.mrj.macos.toolbox.FSSpec;
import com.apple.mrj.macos.generated.FInfoStruct;
import com.apple.mrj.macos.generated.FinderConstants;

public class FileTest
{
    public static void main()
    {
        new FileTest();
    }

    public FileTest()
    {
        //絶対パスを指定してテキストファイルを読む
        //-----------------------------------------------------
        System.out.println("============ Trial 1");
        File targetFile
         = new File("/macos_system/書類/MJR/MJR Samples/ch06_Files/テキストファイル");
            //ファイルの絶対パスの指定方法はこの通り。スラッシュで始まる
            //*********実際に実行する環境のパスに変更してください*******
        try    {
            LineNumberReader inFile = new LineNumberReader(new FileReader(targetFile));
```

```
            //行読み込みを行えるように開く
        String aLine = inFile.readLine();    //1 行分のテキストを読む
        while(aLine != null)     {              //ファイルの最後まで読んだら null を戻す
            System.out.println(aLine);    //読んだテキストをコンソールに出力
            aLine = inFile.readLine();    //1 行分のテキストを読む
        }
        inFile.close();    //ファイルを閉じる
}
catch(Exception e)     {
        System.out.println("Exception:" + e.getMessage());
}

//アプリケーションの存在するフォルダを指定
//-----------------------------------------------------
System.out.println("============ Trial 2");
targetFile = new File("/$APPLICATION/テキストファイル");
        //アプリケーションが存在するフォルダにある「テキストファイル」
        //というファイルを参照する File オブジェクトを生成
File outputFile = new File("/$APPLICATION/テキストファイルの出力");
        //アプリケーションが存在するフォルダにある「テキストファイルの出力」
        //というファイルを参照する File オブジェクトを生成
String lineSeparator = System.getProperty("line.separator");
        //行の区切り文字列をシステムプロパティから取得
try    {
        LineNumberReader inFile = new LineNumberReader(new FileReader(targetFile));
        FileWriter outFile = new FileWriter(outputFile);
            //テキスト出力のためにライターを開く
        int lCounter = 1;  //行数カウンタ
        String aLine = inFile.readLine();    //1 行分のテキストを読む
        while(aLine != null)     {
            System.out.println(aLine);    //コンソールに出力
            outFile.write(String.valueOf(lCounter));  //ファイルに現在の行番号を書き出す
            outFile.write(":");
            outFile.write(aLine);
                //テキストファイルから読み込んだテキストをファイルに書き出す
            outFile.write(lineSeparator);  //改行を適すとファイルに書き出す
            aLine = inFile.readLine();
            lCounter++;        //カウンタを進める
        }
        inFile.close();
        outFile.close();
}
catch(Exception e)     {
        System.out.println("Exception:" + e.getMessage());
}

//作成したファイルのファイルタイプとクリエイターを得る
//-----------------------------------------------------
System.out.println("============ Trial 3");
try    {
        MRJOSType defaultType = MRJFileUtils.getFileType(outputFile);
            //作成したファイルのファイルタイプを得る
        System.out.println(
            outputFile.getName() + "のファイルタイプ：  " + defaultType.toString());
```

```
                    //コンソールに出力
        MRJOSType defaultCreator = MRJFileUtils.getFileCreator(outputFile);
            //作成したファイルのクリエイターを得る
        System.out.println(
            outputFile.getName() + "のクリエイター：  " + defaultCreator.toString());
                //コンソールに出力
}
catch(Exception e)      {
        System.out.println("Exception:" + e.getMessage());
}

//作成したファイルのファイルタイプやクリエイターを設定する
//-----------------------------------------------------
System.out.println("============ Trial 4");
MRJOSType theType = new MRJOSType("TEXT");
                                //付けるファイルタイプはテキストファイル
MRJOSType theCreator = new MRJOSType("ttxt");
                                //付けるクリエイターは SimpleText
try     {
        MRJFileUtils.setFileTypeAndCreator(outputFile, theType, theCreator);
            //ファイルタイプとクリエイターをファイルに一気に設定する
}
catch(Exception e)      {
        System.out.println("Exception:" + e.getMessage());
}

//ファイルタイプやクリエイターの既定値を設定
//-----------------------------------------------------
System.out.println("============ Trial 5");
try     {
        MRJFileUtils.setDefaultFileType(new MRJOSType("TEXT"));
            //既定のファイルタイプとしてテキストファイル
        MRJFileUtils.setDefaultFileCreator(new MRJOSType("JEDT"));
            //既定のクリエイターとして Jedit のクリエイターを設定
        FileWriter outFile
             = new FileWriter("/$APPLICATION/既定値を変えて作ったファイル");
                    //ファイルを新規に作成すると、既定値として
                    //指定したファイルタイプとクリエイターになる
        outFile.close();
}
catch(Exception e)      {
        System.out.println("Exception:" + e.getMessage());
}

//ファイル名一覧からファイル名の扱いを調べる
//-----------------------------------------------------
System.out.println("============ Trial 6");
File appDir = new File("/$APPLICATION");
                                        //アプリケーションが存在するフォルダを参照
String itemList[] = appDir.list(new VisibleFilesOnly());
                                        //表示されているファイル名の一覧を取得する
for(int i=0 ; i<itemList.length; i++)   {       //一覧を順番にコンソールに出力する
        System.out.println(String.valueOf(i) +":"+itemList[i]);
}
```

```
//スラッシュを含むファイル名の指定方法を調べる
//-----------------------------------------------------
System.out.println("============ Trial 7");
String fileName1 = "ファイル名に%2f を含む";
                                        //このようにスラッシュの代わりに%2f と指定
String fileName2 = "ファイル名に/を含む";     //こちらは正しく認識しない
if ((new File("/$APPLICATION/"+ fileName1)).exists())
        System.out.println(" “ " + fileName1 +" ” は存在します。");
else
        System.out.println(" “ " + fileName1 +" ” は存在しません。");
if ((new File("/$APPLICATION/"+ fileName2)).exists())
        System.out.println(" “ " + fileName2 +" ” は存在します。");
else
        System.out.println(" “ " + fileName2 +" ” は存在しません。");

//31 バイトを超えたファイル名を指定してファイルを作ってみる
//-----------------------------------------------------
System.out.println("============ Trial 8");
try     {
        FileWriter outFile
         = new FileWriter(
                "/$APPLICATION/!マックでは許されない長さのファイル名だとどうなる？");
                        //実際にはファイルは作成されるが、最初の 31 バイトだけが
                        //ファイル名として使われる
        outFile.close();
}
catch(Exception e)      {
        System.out.println("Exception:" + e.getMessage());
}

//File Manager の利用
//-----------------------------------------------------
System.out.println("============ Trial 9");

targetFile
 = new File("/macos_system/書類/MJR/MJR Samples/ch06_Files/テキストファイル");
        //存在するファイルのフルパスを指定して、File オブジェクトを作成
        //*********実際に実行する環境のパスに変更してください*******

FSSpec targetSpec = new FSSpec(targetFile);        //File から FSSpec を作成した
        System.out.println(String.valueOf(targetSpec.getVRefNum()));
                                                   //FSSpec の各メンバを見てみる
        System.out.println(String.valueOf(targetSpec.getParID()));
        System.out.println(targetSpec.getName());

FInfoStruct finderInfos = targetSpec.getFInfo();        //Finder 情報を得た
        System.out.println(String.valueOf(finderInfos.getFdFlags()));
                                                   //Finder フラグをコンソールに出力

//File Manager の利用 2（一般的でないと思われる使い方）
//-----------------------------------------------------
System.out.println("============ Trial 10");
        //以下、targetSpec で参照するファイルにロックをかけたりはずしたりする
```

```
        short s;      //Toolbox 関数の戻り値を代入する変数。OSErr 値が戻される

              //以下、ファイル名の文字列を pascal 文字列とした byte 配列を得る
        byte[] fName = targetSpec.getName().getBytes();    //ファイル名の byte 配列を得る
        byte[] fNameP = new byte[fName.length + 1];        //この配列に pascal 文字列を構成する
        fNameP[0] = (byte)fName.length;                    //最初の要素は文字のバイト数
        for( int i = 0 ;   i < fName.length ;   i++)       //2 つ目以降の要素に文字自体を入れる
              fNameP[i+1] = fName[i];

        FSSpecStruct fs = new FSSpecStruct(); //新しく FSSpecStruct を用意
        s = FileFunctions.FSMakeFSSpec(
              targetSpec.getVRefNum(), targetSpec.getParID(),
              fNameP, fs);              //FSSpec が参照するファイルの FSSpecStruct を作成
                    System.out.println(String.valueOf(s));    //戻り値をコンソールへ
        s = FileFunctions.FSpSetFLock(fs);                    //ファイルにロックをかける
                    System.out.println(String.valueOf(s));    //戻り値をコンソールへ
/*      s = FileFunctions.FSpRstFLock(fs);                    //ファイルのロックをはずす
                    System.out.println(String.valueOf(s));    //戻り値をコンソールへ
*/
    }

    class VisibleFilesOnly implements FilenameFilter    //list メソッドで使うフィルタクラス
                        //表示されているファイルだけを取り出し、非表示ファイルや
                        //フォルダは含まれないようにする
    {
        public boolean accept(File dir, String name)
                        //このメソッドが 1 つの項目を取得する度に呼び出される
        {
            File targetFile = new File(dir, name);     //対象ファイルの File オブジェクトを作成
            if (targetFile.isDirectory())    //それがフォルダなら
                return false;                //リストに含めない
            else                             //ファイルなら
            {
                FSSpec targetSpec = new FSSpec(targetFile);
                                             //FSSpec オブジェクトを構築する
                FInfoStruct finderInfos = targetSpec.getFInfo();
                                             //ファイルの Finder 情報を取得する
                if ((finderInfos.getFdFlags() & FinderConstants.kIsInvisible) != 0)
                                             //非表示のフラグをチェック
                    return false;       //フラグがセットされていればリストに含めない
                else
                    return true;
            }
        }
    }
}
```

実行結果

プログラムを実行した結果、コンソールには次の図のように結果が表示されています。また、実行後のアプリケーションの存在しているフォルダには、ファイルが作ら

れています。また、ファイル名の取得などについてのチェックを行うために、記号を含んだファイルなどをあらかじめ作ってあります。

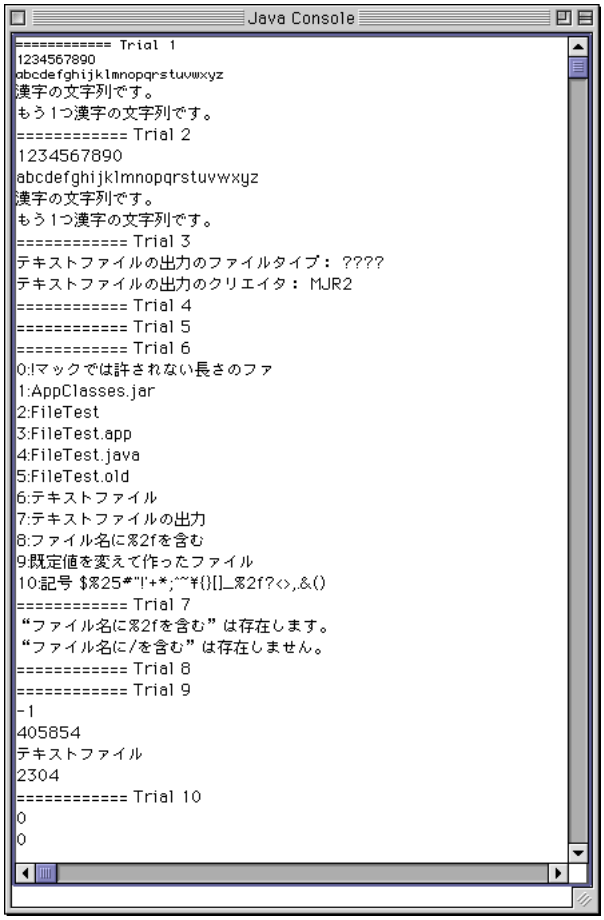

図 6-1　実行後の Java コンソールの内容

図 6-2　実行後のアプリケーションが存在するフォルダ

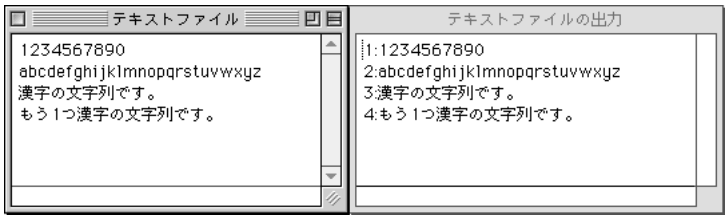

図 6-3　元からあるファイルと実行によって作成されたファイル

第7章　ファイル処理と Mac OS の機能

● アプリケーションファイルをクリエイタで検索したり、システムフォルダなどの特別なフォルダを検索する方法を説明します。

● また、アプリケーションを別プロセスとして起動する方法を説明します。

● エイリアスやプロセスについても説明しますが、執筆段階では確定的な情報がありません。これらに関することは今後状況を見て、更新することにします。

● MRJ 2.1 EA3 での動作をもとに解説します。

7-1 アプリケーションとフォルダの検索

特定のアプリケーションを Java で作ったアプリケーションから起動したい場合、そのアプリケーションのファイルへの参照を、クリエイタから得ることができます。また、「初期設定」フォルダなどシステムが管理する特別なフォルダへの参照も簡単に得られるようになっています。これらは、MRJToolkit の機能として提供されています。

クリエイタからアプリケーションを検索

Java で作ったアプリケーションから、Macintosh にインストールされている別のアプリケーションを起動したいことがあります。Java の基本ライブラリである java.lang.Runtime というクラスの機能を使えば、Mac OS 上の他のアプリケーションを起動することができます。Runtime クラスの使い方は後で説明しましょう。別のアプリケーションを起動するには、そのアプリケーションファイルのパスが必要になります。そのパスを取得する機能は MRJToolkit に組み込まれていますが、以下のようなクラスメソッドを利用します。MRJFileUtils クラスに定義されています。

表 7-1 アプリケーションのファイルを探すメソッド

戻り値	書式	機能
File	MRJFileUtils.findApplication(MRJOSType)	引数に指定したクリエイタを持つアプリケーションファイルを検索し、そのアプリケーションファイルへの参照を File オブジェクトとして得る【Static】

まず、引数には第 6 章で説明した MRJOSType クラスのオブジェクトを指定します。つまり、クリエイタを MRJOSType オブジェクトとして指定します。そうすることで、Mac OS システムはボリュームのデスクトップ情報を検索し、そこからアプリケーションのファイルを取得します。アプリケーション名などからボリュームのファイルを片っ端から検索するわけではなく、Mac OS のデスクトップの機能を利用しているため、検索は非常に高速に行われます。

戻り値は java.io.File クラスのオブジェクトです。File オブジェクトの扱いは第 6 章で解説したように、一般的な Java でのファイルの扱いが可能となるわけです。

なお、findApplication メソッドで指定したクリエイタに対するアプリケーションが見つからない場合、例外が発生します。FileNotFoundException というクラスの例外が発生するので、この例外をハンドリングするようにプログラムを作成しておく必要があります。

フォルダの検索

Mac OS で稼動するアプリケーションの場合、システムフォルダ内の「初期設定」フォルダに、各種の設定を残したいと考えるかもしれません。また、「アプリケーションサポート」に情報を残したり、あるいはアップルメニューの項目を自分で作るようなニーズがあるかも知れません。そのようなときに、フォルダへの参照を得るための機能が MRJToolkit には用意されています。以下の、MRJFileUtils クラスのクラスメソッドを利用して、フォルダへの参照を File オブジェクトとして得ることができます。

表 7-2　特別なフォルダを探すメソッド

戻り値	書式	機能
File	MRJFileUtils.findFolder(MRJOSType)	フォルダへの参照を File オブジェクトとして得る。引数に、検索するフォルダを示す記号を指定するが、MRJFileUtils クラスのクラス変数を利用するのが一般的【Static】

findFolder によってフォルダが見つからない場合には、FileNotFoundException が発生します。この例外処理をハンドリングしておく必要があります。

ここで、引数に指定できる定数は表 7-3 にまとめておきました。これらは static final で定義されたクラス変数ですので、たとえば次のようにして指定します。例外処理を含むプログラムは、この章の最後の節を参照してください。

```
import java.io.*;
import com.apple.mrj.MRJOSType;
import com.apple.mrj.MRJFileUtils;

    :

File prefFolder;
prefFolder = MRJFileUtils.findFolder(MRJFileUtils.kPreferencesFolderType);
                    //「初期設定」フォルダへの参照を得る
```

これにより変数 prefFolder は、アプリケーションが稼動している Mac OS システム

の「初期設定」フォルダを参照します。以後、必要に応じてそこにあるファイルを開いたり、あるいはファイルへ保存するということを行います。

なお、表 7-3 で紹介したクラス変数はすべてが利用できるわけではないようです。MRJ の EA 版のせいかもしれませんが、一部のクラス変数は指定できませんでした。たとえば、kModemScriptsFolderType を指定しても、コンパイル時に定義されていないとエラーが出ます。これは、MRJClasses.zip ファイルが最新の状況を反映していないためではないかと想像できますが、その場合は値を直接指定することで、プログラムは機能させることができます。たとえば次の様なプログラムになります。

```
prefFolder = MRJFileUtils.findFolder(new MRJOSType(0xC46D6F64));
//kModemScriptsFolderType フォルダへの参照を得る
```

kModemScriptsFolderType は、値としては 'ト mod' と定義されています。その int での記述方法は、MRJ SDK の JDirect フォルダの JDirect 2.0 Sample Code フォルダにある Folders.java というソースファイルに記述されています。それを自分で作っているプログラムのソースにコピーすればいいでしょう。

表 7-3　findFolder の引数に指定するクラス変数

定数	Static 変数	検索されるフォルダ
'macs'	kSystemFolderType	システムフォルダ
'desk'	kDesktopFolderType	デスクトップフォルダ
'trsh'	kTrashFolderType	ゴミ箱
'empt	kWhereToEmptyTrashFolderType	ネットワークで共有しているゴミ箱フォルダ
'prnt'	kPrintMonitorDocsFolderType	「プリントモニタ書類」フォルダ（システムフォルダのルート）
'strt'	kStartupFolderType	「起動項目」フォルダ（システムフォルダのルート）
'shdf'	kShutdownFolderType	「システム終了項目」フォルダ（システムフォルダのルート）
'amnu'	kAppleMenuFolderType	「アップルメニュー」フォルダ（システムフォルダのルート）
'ctrl'	kControlPanelFolderType	「コントロールパネル」フォルダ（システムフォルダのルート）
'extn'	kExtensionFolderType	「機能拡張」フォルダ（システムフォルダのルート）
'font'	kFontsFolderType	「フォント」フォルダ（システムフォルダのルート）
'pref'	kPreferencesFolderType	「初期設定」フォルダ（システムフォルダのルート）
'temp'	kTemporaryFolderType	テンポラリフォルダ（非表示）

定数	Static 変数	検索されるフォルダ
'extD'	kExtensionDisabledFolderType	「機能拡張（使用停止）」フォルダ（システムフォルダのルート）
'ctrD'	kControlPanelDisabledFolderType	「コントロールパネル（使用停止）」フォルダ（システムフォルダのルート）
'macD'	kSystemExtensionDisabledFolderType	「システム機能拡張（使用停止）」フォルダ（システムフォルダのルート）
'strD'	kStartupItemsDisabledFolderType	「起動項目（使用停止）」フォルダ（システムフォルダのルート）
'shdD'	kShutdownItemsDisabledFolderType	「システム終了項目（使用停止）」フォルダ（システムフォルダのルート）
'apps'	kApplicationsFolderType	「アプリケーション」フォルダ
'docs'	kDocumentsFolderType	「書類」フォルダ
'root'	kVolumeRootFolderType	ボリュームのルート
'flnt'	kChewableItemsFolderType	起動時に削除される項目を入れるフォルダ
'asup'	kApplicationSupportFolderType	「アプリケーションサポート」フォルダ（システムフォルダのルート）
'ト tex'	kTextEncodingsFolderType	「テキストエンコーディング」フォルダ（システムフォルダのルート）
'odst'	kStationeryFolderType	「OpenDoc ひな形」フォルダ
'odod'	kOpenDocFolderType	OpenDoc フォルダ（「エディタ」フォルダ）
'odsp'	kOpenDocShellPlugInsFolderType	「OpenDoc Shell プラグイン」フォルダ（「OpenDoc」フォルダ）
'oded'	kEditorsFolderType	「エディタ」フォルダ（システムフォルダのルート）
'ト odf'	kOpenDocEditorsFolderType	「OpenDoc 基本キット」フォルダ？（「エディタ」フォルダ）
'odlb'	kOpenDocLibrariesFolderType	「OpenDoc ライブラリ」フォルダ（「エディタ」フォルダ）
'ト edi'	kGenEditorsFolderType	システムフォルダの General Editors フォルダ？（エラーになる）
'ト hlp	kHelpFolderType	「ヘルプ」フォルダ（システムフォルダのルート）
'ト net'	kInternetPlugInFolderType	プラグインのフォルダ
'ト mod'	kModemScriptsFolderType	Modem Scripts フォルダ（「機能拡張」フォルダ）
'ppdf'	kPrinterDescriptionFolderType	「プリンタ記述ファイル」フォルダ（「機能拡張」フォルダ）
'ト prd'	kPrinterDriverFolderType	プリンタドライバ向けのシステムフォルダに作られる新しいフォルダ
'ト scr'	kScriptingAdditionsFolderType	「スクリプティング機能追加」フォルダ（システムフォルダのルート）
'ト lib'	kSharedLibrariesFolderType	共有ライブラリのフォルダ
'fvoc'	kVoicesFolderType	Voice フォルダ（MacinTalk で使用）

定数	Static 変数	検索されるフォルダ
'sdev'	kControlStripModulesFolderType	「コントロールバー項目」フォルダ（システムフォルダのルート）
'astト'	kAssistantsFolderType	「アシスタント」フォルダ（ボリュームのルート）
'utiト'	kUtilitiesFolderType	「ユーティリティ」フォルダ（ボリュームのルート）
'aexト'	kAppleExtrasFolderType	「Apple エクストラ」フォルダ（ボリュームのルート）
'cmnu'	kContextualMenuItemsFolderType	「コンテクストメニュー項目」フォルダ（システムフォルダのルート）
'morト'	kMacOSReadMesFolderType	「Mac OS 情報」フォルダ（ボリュームのルート）
'walk'	kALMModulesFolderType	「作業環境マネージャモジュール」フォルダ（「機能拡張」フォルダ）
'trip'	kALMPreferencesFolderType	「作業環境マネージャ初期設定」フォルダ（「初期設定」フォルダ）
'fall'	kALMLocationsFolderType	「作業環境」フォルダ（「作業環境マネージャ初期設定」フォルダ）
'prof'	kColorSyncProfilesFolderType	「ColorSync 特性」フォルダ（システムフォルダのルート）
'appr'	kAppearanceFolderType	「アピアランス」フォルダ（システムフォルダのルート）
'snds'	kSoundSetsFolderType	「サウンドセット」フォルダ（「アピアランス」フォルダ）
'dtpト'	kDesktopPicturesFolderType	「デスクトップピクチャ」フォルダ（「アピアランス」フォルダ）
'thme'	kThemesFolderType	「テーマファイル」フォルダ（「アピアランス」フォルダ）
'favs'	kFavoritesFolderType	「よく使う項目」フォルダ（システムフォルダのルート）
'intト'	kInternetFolderType	「インターネット」フォルダ（ボリュームのルート）
'issf'	kInternetSearchSitesFolderType	「インターネット検索サイト」フォルダ（システムフォルダのルート）
'fnds'	kFindSupportFolderType	「検索」フォルダ（「機能拡張」フォルダ）
'fbcf'	kFindByContentFolderType	TheFindByContentFolder。ボリュームのルートに作られる非表示フォルダ
'ilgf'<	kInstallerLogsFolderType	インストーラーのログフォルダ
'scrト'	kScriptsFolderType	「スクリプト」フォルダ（システムフォルダのルート）
'fasf'	kFolderActionsFolderType	Folder Actions Scripts フォルダ？
'laun'	kLauncherItemsFolderType	「ランチャー項目」フォルダ（システムフォルダのルート）
'rapp'	kRecentApplicationsFolderType	「最近使ったアプリケーション」フォルダ（「アップルメニュー」フォルダ）

定数	Static 変数	検索されるフォルダ
'rdoc'	kRecentDocumentsFolderType	「最近使った書類」フォルダ（「アップルメニュー」フォルダ）
'rsvr'	kRecentServersFolderType	「最近使ったサーバ」フォルダ（「アップルメニュー」フォルダ）
'spki'	kSpeakableItemsFolderType	「音声」フォルダ（システムフォルダのルート）

7-2 エイリアスの取り扱い

エイリアスを扱う機能が MRJ には含まれているようですが、MRJ 2.1EA3 では、MRJDeprecatedClasses.zip というファイルにだけあり、現状では利用できず、再構築中ではないかと思われます。具体的な使用方法は、MRJ のバージョンが上がってから紹介します。

エイリアスを扱うクラス

MRJ にはエイリアスを扱う com.apple.MacOS.Alias というクラスが定義されていますが、MRJDeprecatedClasses.zip というファイルにコンパイル結果がおさめられています。このファイルをプロジェクトに登録してコンパイルできなくはないのですが、Alias クラスの機能に関してはデバッガに落ちてしまい、実際に使うことができませんでした。おそらく、きちんとした内容のクラスを構築中なのではないかと想像できます。現状でわかる範囲でどんな様子なのかをいちおうまとめておきます。

この Alias クラスにはコンストラクタがあり、FSSpec、File、String を引数にしてオブジェクトを生成できるようです。引数に既存のファイルやフォルダを指定して、それへのエイリアス情報をメモリ上に作るのだと思われます。そして、toSpec というメソッドが定義されており、Alias オブジェクトから、FSSpec オブジェクトを得るようになっています。Toolbox の API コールで言えば、コンストラクタが NewAlias、toSpec メソッドは ResolveAlias に相当するものだと考えられます。

つまりエイリアスレコードは作成でき、それから元のファイルを検索できるようにはなっているのです。しかしながら、この Alias オブジェクトをファイルなどに保存ができないと応用上はどうしようもないところだと思います。期待するのは、たとえば Alias クラス自体が Serializable で定義されていて、オブジェクト自体を簡単にファイルなどに保存し復元できるというところです。しかしながら、Serializable ではなく、また、クラスのフィールド変数へもアクセスできないようです。つまり、現状のままでは仮に利用できたとしも、Alias オブジェクトの保存のしようがない状態と言えます。

なんらかの理由でファイルのありかを記録したい場合、ファイルの絶対パスなどの文字列を記録しておくというが代表的な方法です。ただし、そうなると、そのファイルを後でユーザーが異なるフォルダに移動したときに見つけられなくなります。そこ

で、エイリアスによって記録しておけば、移動後やあるいはファイル名変更があっても元のファイルを見つけることができるようになります。しかしながら、そこまでの機能を要求されないのであれば、パス名の記録でも十分だと言えるでしょう。現状ではエイリアスが使いにくいこともあるのですが、用途に応じた機能を使うということは言うまでもありません。

エイリアスファイルの元ファイルを求める

エイリアスを扱う別の目的として、エイリアスファイルから、そのエイリアスのオリジナルファイルを求めるということがあります。そのためには ResolveAliasFile という Toolbox の API を利用します。この API コールも Java で利用できるようになっており、com.apple.mrj.macos.generated.AliasFunctions というクラスで定義されています。元ファイルは FSSpecStruct で指定するのですが、それ以外にいくつか引数を指定することになっています。その引数に関する情報が得られないことや、byte 型の配列になっていることを考えると、この ResolveAliasFiles メソッドは基本的には内部的に使うものではないかと考えられます。

Alias クラスが再構築されたときには、その中のクラスメソッドとして ResolveAliasFile も利用できるようになるのではないかと考えられます。ここでは、現状の ResolveAliasFile の利用方法までは追求しないことにします。

7-3 プロセスの取り扱い

Mac OS での起動した 1 つのアプリケーションを、Java では「プロセス」と呼びます。こうした OS ごとに異なる事情があるとはいえ、複数のプロセスを同時に起動して切り替えることができるというのはどの OS も同様です。java.lang パッケージに用意された汎用のプロセス処理と、Mac OS 向けのプロセス処理をうまく使う必要がありますが、Mac OS 向けの処理はそれほど豊富ではありません。つまり、プロセス処理は基本的なことしかできないと考えておいて良いでしょう。

Runtime クラス

OS のシステム機能を利用するためのクラスとして、java.lang.Runtime が基本ライブラリのパッケージに用意されています。この Runtime のオブジェクトを参照すれば機能を利用できるのですが、オブジェクトを取得するクラスメソッドを使うのが一般的です。このクラスで利用できる代表的な機能は以下の通りです。基本的には、getRuntime で Runtime オブジェクトへの参照を得て、他のメソッドを利用するというのが手順となります。

表 7-4 Runtime クラスの代表的なメソッド

戻り値	メソッドの書式	機能
Runtime	Runtime.getRuntime()	Runtime オブジェクトを取得する【Static】
Process	exec(String[])	別のプロセスを起動する。引数にアプリケーションや開くファイルを指定する（別項を参照）
void	exit(int)	現在のプロセスを終了する。つまり、Java のアプリケーションを終了する。引数には OS に返すエラーコードを指定するが、Mac OS では無視されるので何を指定してもよい
long	freeMemory()	現在の空きメモリのバイト数
long	totalMemory()	利用できるメモリのバイト数
void	gc()	ガベージコレクションを行う

♦アプリケーションの起動

Runtime クラスの exec メソッドを使えば、別のプロセスを起動することができます。

つまり、Mac OS では別のアプリケーションの起動ができます。ここで、exec にはいくつかの引数のバリエーションがあるのですが、MRJ で利用できるのは、String オブジェクトの配列を指定したときだけで、その他の場合にはエラーが出ます。

引数に指定する配列の最初の要素（つまり 0 番目の要素）には、起動するアプリケーションのファイルを指定します。基本的には絶対パスを指定するのが良いでしょう。アプリケーションのファイルは、MRJFileUtils クラスのクラスメソッドである findApplication で得られます。その得られた File オブジェクトから、getAbsolutePath メソッドを使うなどして、アプリケーションファイルの絶対パスを得て、それを指定するのが確実です。

配列の要素が 1 つだけなら、そのアプリケーションを起動します。配列の要素が 2 つ以上なら、アプリケーションを起動して、2 つ目以降の要素に指定したファイルを開きます。つまり、ファイルをアプリケーションにドラッグ&ドロップして起動したのと同じことになります。このとき、2 つ目以降の要素（つまり、1 番目以降の要素）には、開くファイルの絶対パスを指定します。やはり、getAbsolutePath メソッドを使うのが基本でしょう。たとえば、プログラム例としては以下のようになります。

```
import java.io.*;
import com.apple.mrj.MRJOSType;
import com.apple.mrj.MRJFileUtils;

     :
File appFile;//アプリケーションのファイルを参照する変数
File docFile1, docFile2;
try{
    appFile = MRJFileUtils.findApplication(new MRJOSType("ttxt"));
          //ファイルタイプが ttxt のアプリケーション（SimpleText）のファイルを取得
}
catch(Exception e){System.out.println(e.getMessage());}
//例外処理

docFile1 = new File("/$APPLICATION/FolderTest.java");
docFile2 = new File("/$APPLICATION/TrivialApplication.java");
    //このアプリケーションと同じフォルダにある 2 つのファイルを参照する

String params[] = new String[3];              //コマンドラインの文字列
params[0] = appFile.getAbsolutePath();        //最初は起動するアプリケーション
params[1] = docFile1.getAbsolutePath();       //2 つ目以降は開くファイル
params[2] = docFile2.getAbsolutePath();
stProccess = (MacOSProcess)Runtime.getRuntime().exec(params);
                                              //SimpleText を起動してファイルを開く
```

起動プロセスの取り扱い

Runtime クラスの exec でアプリケーションを起動すると、その起動したプロセスを参照するオブジェクトが戻されます。定義上は Process クラスですが、これは、java.lang パッケージに定義されたものです。ただし、Process クラスは Abstruct クラスとして定義されており、実際には実行している OS ごとに、Process のサブクラスを定義することになっています。MRJ では、sun.misc.MacOSProcess というクラスが定義されていて、exec メソッドの戻り値は MacOSProcess クラスのオブジェクトになっています。

ただし Process と MacOSProcess の違いは、メンバ変数として、psn というものが MacOSProcess に追加されているだけで、メソッドはまったく同様です。つまり、MacOSProcess に Mac OS 独自の機能が加わっているわけではありません。psn というメンバ変数は public なので、プログラム上で取得することができます。おそらく、Toolbox でのプロセスシリアルナンバーだと思われます。

Process あるいは MacOSProcess に用意されているメソッドの中で実際に使えるのは、プロセスを終了、つまりアプリケーションを終了させる destroy メソッドだけでしょう。

プロセスシリアルナンバーの処理

Toolbox の Process Manager では、Mac OS での起動アプリケーションであるプロセスを管理するさまざまな機能を提供しています。これと同等なことができるように、MRJ でクラスが作られていればいいのですが、実質的にはそこまでは行われていません。

たとえば、起動しているプロセスの一覧を得て、各プロセスの詳細な情報を得るというためのクラス定義がならあります。com.apple.mrj.macos.toolbox.ProcessSerialNumber クラスや、com.apple.mrj.macos.toolbox.ProcessInfoRec クラスがそれに相当するものです。ProcessSerialNumber クラスにプロセスを取得する機能があるようですが、試した限りではエラーなく実行できたことはありませんでした。プロセス情報を取得するところで必ずエラーになってしまいます。

一方、Process Manager の機能をそのまま使える ProcessFunctions クラスも定義されているようですが、データの扱いが文字列データでもおそらくポインタの値として扱うような形態のため、手軽に利用することは難しいと思われます。

MRJ の Process Manager 関連の機能としては、残念ながら動くサンプルが作成でき

なかったため、詳細をお届けることは当面は見送ることにしました。

7-4　サンプルプログラム

第 7 章に説明したことを含むサンプルプログラムです。アプリケーションの起動や終了、ファイルを開くこと、フォルダの検索が含まれています。

FolderTest.java

```
import java.io.*;
import com.apple.mrj.MRJOSType;
import com.apple.mrj.MRJFileUtils;
￥

public class FolderTest
{
    public static void main(){
        new FolderTest();
    }

    public FolderTest(){

    File appFile;//アプリケーションのファイルを参照する変数
    try{
        appFile = MRJFileUtils.findApplication(new MRJOSType("MSIE"));
            //ファイルタイプが MSIE のアプリケーション(Internet Explorer)のファイルを取得
        System.out.println(appFile.getAbsolutePath());
            //得られたファイルのフルパスをコンソールに出力

        String params[] = { appFile.getAbsolutePath() };
            //MSIE のファイルのフルパスを要素に持つ文字列の配列を作成
        Process p = Runtime.getRuntime().exec(params);   //Internet Explorer を起動する
        System.out.println(p.getClass().getName());   //Process の実際のクラスを調べた
        p.destroy();//起動した Internet Explorer を終了する
    }
    catch(Exception e){System.out.println(e.getMessage());}
        //例外処理

    File docFile1, docFile2;
    try{
        appFile = MRJFileUtils.findApplication(new MRJOSType("ttxt"));
            //ファイルタイプが ttxt のアプリケーション(SimpleText)のファイルを取得
        System.out.println(appFile.getAbsolutePath());
            //得られたファイルのフルパスをコンソールに出力

        docFile1 = new File("/$APPLICATION/FolderTest.java");
        docFile2 = new File("/$APPLICATION/TrivialApplication.java");
            //このアプリケーションと同じフォルダにある 2 つのファイルを参照する

        String params[] = new String[3];//コマンドラインの文字列
```

```
            params[0] = appFile.getAbsolutePath();//最初は起動するアプリケーション
            params[1] = docFile1.getAbsolutePath();//2 つ目以降は開くファイル
            params[2] = docFile2.getAbsolutePath();
            stProccess = (MacOSProcess)Runtime.getRuntime().exec(params);
                                                    //SimpleText を起動してファイルを開く
        }
        catch(Exception e){System.out.println(e.getMessage());}
            //例外処理

        //Runtime クラスの情報を取得してみた
        System.out.println(String.valueOf(Runtime.getRuntime().freeMemory()));
        System.out.println(String.valueOf(Runtime.getRuntime().totalMemory()));

        File prefFolder;
        try{
            prefFolder = MRJFileUtils.findFolder(MRJFileUtils.kPreferencesFolderType);
                //「初期設定」フォルダへの参照を得る
            System.out.println(prefFolder.getAbsolutePath());
                //得られたフォルダのフルパスをコンソールに出力
        }
        catch(Exception e){System.out.println(e.getMessage());}
            //例外処理
        }
}
```

LOCUS Report

第8章　Mac OS で使う Swing

● 機能豊富な GUI コンポーネントを利用できる Swing は、Java で手軽に高機能なソフトウエアを作成することにもつながり大変に注目されています。

● Swing は Java2 に標準で組み込まれました。しかしながら、JDK1.1.X 上でもライブラリを追加することで利用できます。

● もちろん、Mac OS 上でも利用できます。MRJ 2.1 を利用した上で Swing を利用したプログラム作成について説明をしましょう。

● Swing は、ある意味では基本コンポーネントである AWT の拡張です。この章では、AWT でのプログラミングとの違い、そして Mac OS で動かすプログラムでの注意点を中心に解説を行います。

8-1　Mac OS で使う Swing

Swing についての基本的なことと、Mac OS で Swing を使って開発するために必要な作業についてまとめておきます。

Swing の特徴

ウインドウやメニューのような GUI に欠かせないものに加え、テキストボックスやボタンなどを始めとするコントロール類もよく利用されます。古くはそうしたコントロールを独自に設計して作り、それを利用するような時代もありました。しかしながら、現在はシステム機能としてコントロールを豊富にサポートし、プログラムをする側は、手軽にたくさんのコントロールを使うということを要求するようになっています。Windows の Visual Basic のように、最初からたくさんのコントロールが利用でき、さらに追加することもできるため、コントロール類だけを販売するようなビジネスまで成り立つようになってきています。Mac OS では、Ver.8.0 で、コントロールの種類を大幅に増やし、かなりのバリエーションのコントロールが、指定するだけで利用できるようになりました。

こうしたコントロールを使うメリットは、もちろん、手軽にプログラム作成できるという点にあります。豊富なコントロールがあれば、手軽に高い機能のソフトウエアを作ることにもつながります。コントロールは GUI 部品に限りませんが、GUI 部品として使われるものが多くなっています。

Java の基本ライブラリでは、AWT の中で、コントロールを当初からサポートしてきていました。しかしながら、利用できるコントロールは基本的なものに限られていました。そこで、JFC（Java Foundation Class）として、豊富なコントロールをサポートするようになり、JDK 1.1.X の時代に開発が進められてきました。Java2 では JFC は標準コンポーネントとなっています。Swing は JFC のもっとも重要な要素で、豊富な GUI 部品を提供するライブラリです。JFC で提供される大部分は Swing であるとも言って良いくらいで、注目が集まったためにコードネームとしての Swing が正式名称となってしまったと言ってよいでしょう。

♦機能が豊富な GUI コンポーネント

Swing の注目できるところは機能が豊富なところで、つまり多種多様なコントロールがサポートされているということでしょう。AWT までしか使えなかった時代では、

Java の機能不足が何かにつけて取りざたされていましたが、基本的なコントロールしか使えない状態ではそれも否定できない状態でした。しかしながら、Swing はそれを打破したと言えるでしょう。現在、一般的なアプリケーションに使われているようなさまざまなコントロールは、ほとんどが Swing によってプログラムで利用できるようになります。いずれにしても、Java で作るプログラムのユーザーインタフェースを多彩にする極めて重要なコンポーネントが Swing であると言えるでしょう。

用意されているコンポーネントは、非常にたくさんあります。目に付く点としては、まずツールバーが簡単に構築できるという点があります。また、メニューやボタンは非常に多くの表示バリエーションを持っています。テキスト関連のコントロールでは、単なるテキストエディタだけでなく、スタイル付きのエディタやその発展系として HTML のエディタまであります。もちろん、単にコンポーネントをウインドウに貼り付ければエディタを作ることができるというわけではないものの、表示するだけであれば、コンポーネントの配置だけで可能でもあり、その意味でも豊富で高い機能の一端が表現できるのではないでしょうか。表形式のテーブルコントロールも利用できます。また、テキストだけでなく多くのコントロールでグラフィックスも利用できるなど、見栄えのするユーザーインタフェースを作成するというレベルまでの機能を持っています。

Swing はコントロール群というだけでなく、ユニークな機能も見られます。まず、ルック&フィールとしてコンポーネントの見かけやあるいは諸設定をクラスとして定義することが挙げられます。これにより、ソフトウエア全体の見かけを、使用するクラスの取り替えで簡単に変えられるということになります。標準的なルック&フィールが用意されていますが、Windows 風、Macintosh 風などが用意されており、OS の使用感に近付けることもできます。また、独自のカスタマイズもできると言えるでしょう。つまり、Mac OS で言えばアピアランスあるいはテーマのような機能が利用できるわけです。

♦Java で構築されているライブラリ

Swing 自体は、すべての部分が Java で作られているため、ライブラリ自体はプラットフォームに依存するものではありません。つまり、Java が稼動していれば、Swing が必ず利用できると言えるわけです。AWT を基礎にして、Java だけで構築しているため、Swing 自体のクロスプラットフォームの問題は比較的出にくい状況にあるとも言えるでしょう。

実用面から見た Swing

機能が高くなれば、重くなることは避けられません。実用面でどうなのかを検討してみましょう。

まず、Swing のライブラリ自体のファイルは 2MB 程度で、そこそこのサイズです。アプリケーションといっしょに配付するとなると、ちょっと大きなサイズであるかなとも思えるかもしれませんが、MRJ の場合、Java 標準クラスだけでも 7MB を超えるサイズであることを考えれば、むしろこれほどの機能の割には小さいとも言えるでしょう。また、バージョンアップはあるものの、Java2 では標準に組み込まれる機能なので、配付についての心配も、特に将来的には少なくなると思われます。

Swing の動作については、やはりただでさえ遅いことが指摘される Java の上で、その Java を使って作られたライブラリだけに、やはり重たい物になっています。もちろん、ネイティブな機能を使ってシャキシャキ動く GUI ライブラリという方向性も短期的には実用的かも知れませんが、こうしたライブラリ自体を Java で作るというのが、Java の 1 つの進む道筋を表しているとも言えます。マシンの性能が加速度的に向上している現在では、重いと思っていたソフトもいつの間にか軽くなっていることもあります。少し先を見た場合には、重たいという問題の比重も低くなるのではないかと思います。

メモリについてもかなり食います。Mac OS で、1M 程度のアプリケーションメモリを確保していても、Swing を使うとそのアプリケーションメモリが何と 10M くらいまで一気に広がります。メモリ自体のコストも安くなったとは言え、概してブラウザ並に 1 つのアプリケーションがメモリを消費するというのは、現状のアプリケーションの水準からしてかなり多くのメモリを使うと言わざるを得ないでしょう。配付して使うようなアプリケーションの場合には、それなりに配慮は必要なレベルだと思われます。

また、Mac OS 環境ということから見た場合、完全に OS 機能と統合されていない面はあります。具体的に次の節で説明をしますが、Swing はまだ発展途上にあると言えるような段階です。もちろん、Ver.1.1 まで正式版が出ており、それなりに完成度は高くなっていますが、概して、完成したと結論付けるにはまだ早い段階だと思います。もういくらかもまれる必要があると考えられます。

Mac OS 環境では、MRJ に Swing が組み込まれていないこともあり、Swing によるソフト開発は、状況にもよりますが、まだ時期尚早な面もなくはないでしょう。すぐ

に利用したい実用ソフトでは、スペックの低いマシンで限定された場合などは利用をあきらめるという場面も出てくると思われます。一方、将来を見越した場合には、Java の世界でも必須の機能だと言えます。マシン環境が良くなるなどの環境の変化を考えれば、Swing は将来的には大きく有望な機能であることは間違いないことと言えるでしょう。

Swing ライブラリの組み込み

Mac OS で Swing 対応のアプリケーションなどを開発する場合に必要な準備を説明しましょう。まず、ここでは、MRJ 2.1 および MRJ SDK 2.1 の正式リリース版をインストールしているものとします。

MRJ 2.1 は、対応する JDK のバージョンが 1.1.6 であり、Swing を含まない Java の実行環境です。そのため、Swing を別途入手する必要があります。Swing 自体は販売されているわけでなく、以下のアドレスより無償で入手できます。

http://java.sun.com/products/jfc/

MRJ 2.1 上で開発するときには、JDK 1.1 対応の JFC を入手します。1999 年 3 月末のこの原稿を執筆している段階では、Swing 1.03、Swing 1.1、Swing 1.1.1β1 の 3 種類の JFC がリリースされていました。ともかくいちばん新しい、Swing 1.1.1β1 を使ってサンプルプログラムを作成しました。

ダウンロードは上記の Web ページで操作をして行いますが、使用する OS を選択するようになっています。ここで、Mac OS で開発するのでれば、Mac OS 向けのものをダウンロードすると良いでしょう。すると、Mac OS のアプリケーション形式のインストーラがダウンロードできます。開発キットと言えば、ファイルを圧縮しただけのようなものがむしろ多いのですが、MRJ SDK などと同様、インストーラでインストールができます。ただし、基本的にはファイルを展開するだけのようです。ちなみに、インストーラは InstallAnywhere を使って作られたもののようです。

図 8-1　ダウンロードしたインストーラ

インストーラの利用方法は難しいことはありません。単にダブルクリックして、以下のようなウィザード形式の質問に答えて行けばそれでファイルがハードディスクに展開されます。以下、インストーラのポイントになる部分だけをかいつまんで説明をしておきましょう。

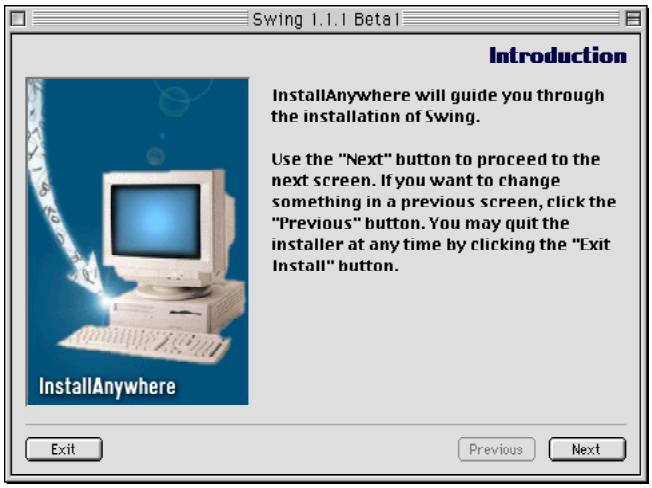

図 8-2　ウィザード形式のインストーラ

インストール作業の途中で、次のように、インストール先のフォルダを指定する画面になります。ここで、Choose を選択して、ファイルを選んでもかまいません。既定値では「アプリケーション」フォルダになっています。日本語のフォルダ名を選択しても機能します。

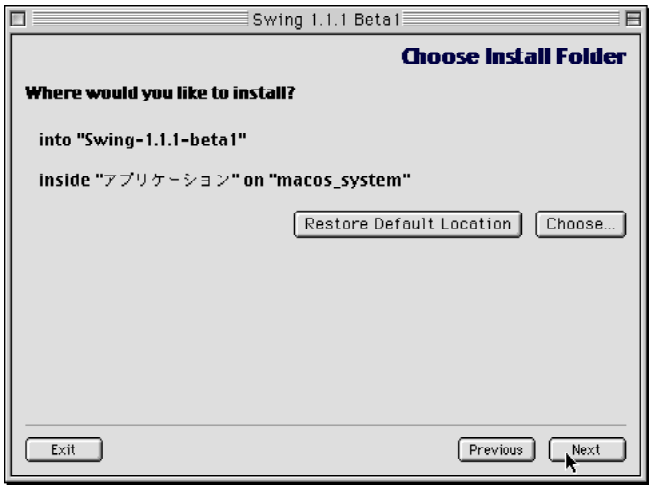

図 8-3　保存する先を指定する

インストーラの途中では次のように、エイリアスをどこに作成するかを選択する画面になります。この中に、何のエイリアスなのかが明確に書かれていないので分かりづらいのですが、これは、Swing のサンプルプログラムのアプリケーションへのエイリアスです。デスクトップにサンプルのエイリアスがあると、もちろん、サンプルを即座に試すことができるので便利と言えば便利ですが、サンプルをしょっちゅう起動するということもないと思われるので、不要なら Don't create aliases を選択しておくとよいでしょう。

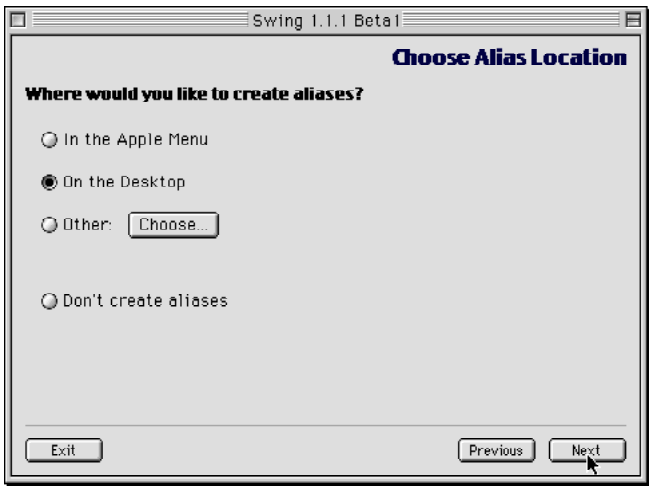

図 8-4　サンプルアプリケーションのエイリアスのインストール先を指定

さらに、次のように、セットアップする内容を選択する画面になります。開発を行うのであれば、通常はいちばん下の Full Developer Release を選択することになるでしょう。クラスライブラリの使い方を記したドキュメントは、これを選択しないとイン

ストールされません。

図 8-5　インストールする内容は、Full Developer Release を選択する

こうしてインストールが完了します。まず、サンプルアプリケーションへのエイリアスをデスクトップに作成するように指定したので、次の図のように、アプリケーションのデフォルトアイコンで、アプリケーションへのエイリアスが作られています。SwingSet というのが Swing のコンポーネントを一覧にして見せるアプリケーションで、よく引き合いに出されるものです。アイコンはデフォルトになっていますが、ダブルクリックすると起動して更新され、Java のキャラクターがデザインされたアイコンに変わります。

図 8-6　デフォルトのアプリケーションアイコンがデスクトップに配置されたサンプルのエイリアス

ここで、実際に Swing 対応のアプリケーションが機能するかをチェックすれば良いでしょう。SwingSet というエイリアスをダブルクリックすると、Swing のコンポーネントにどのようなものがあって、Mac OS では実際にどんなふうに見えるかがわかるので、これを実行してみれば良いでしょう。

また、インストール先のフォルダとしてここでは「アプリケーション」フォルダを選択しましたが、インストール先には「Swing-1.1.1-beta1」というフォルダが作られていて、その中に、次のようにファイルが展開されています。これらのファイルを開発あるいは実行時に利用します。

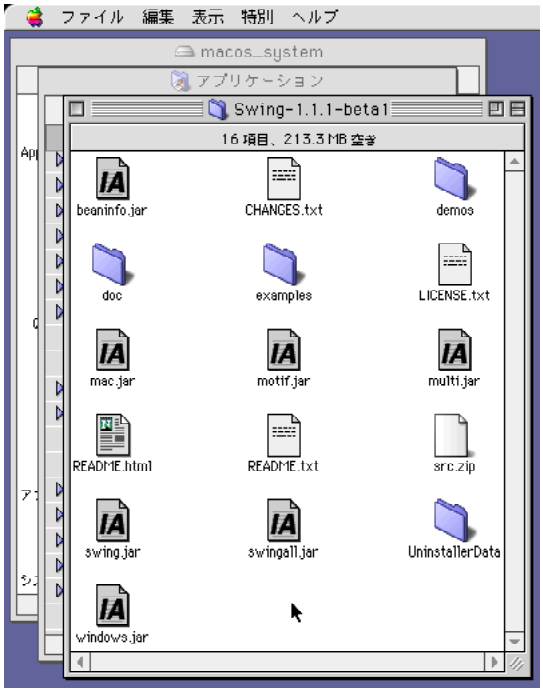

図 8-7　インストールされたフォルダの中身

ここで、フォルダの中にいくつかファイルが展開されていますが、拡張子が.jar のいくつかあるファイルが Swing の本体です。通常は Swingall.jar というファイルを利用します。使用するファイルについてはこの後で紹介します。なお、いずれのファイルもアーカイブされたままの状態で使用します。

表 8-1　インストールによって得られるライブラリファイル

ファイル名	内容
swing.jar	Swing のクラスライブラリ本体。ルック&フィールは Metal だけが含まれる
windows.jar	Windows のルック&フィールを納めたファイル

motif.jar	Motif のルック&フィールを納めたファイル
beaninfo.jar	Beans に関連するファイル
swingall.jar	swing.jar, windows.jar, motif.jar, beaninfo.jar の内容を 1 つにまとめたファイル
mac.jar	Mac のルック&フィールを納めたファイル
multi.jar	ルック&フィールを複数利用するための機能を提供するファイル

あと、README.html が Netscape Navigator の文書で含まれています。テキストファイルが Mac OS の改行形式になっていないなどの理由から、添付文書はいずれもブラウザで見る事になります。README.html を開けば、基本的な注意書きだけでなく、フォルダに含まれる各種文書へのリンクもあります。また、ライブラリの使い方のドキュメントへのリンクもあります。

プログラムで Swing を利用する

Swing を利用したプログラムを CodeWarrior で作る場合の一般的な手順を示しておきましょう。個別の内容として次の節で説明することもありますが、ここでは一般的な内容を説明しておきます。

まず、Java のアプリケーションあるいはアプレットを作成するためのプロジェクトを用意します。Java のカテゴリにあるいずれかのプロジェクトのひな形を選択して、保存するフォルダを指定するのは通常通りです。以下は、Java Application を指定した場合を説明します。

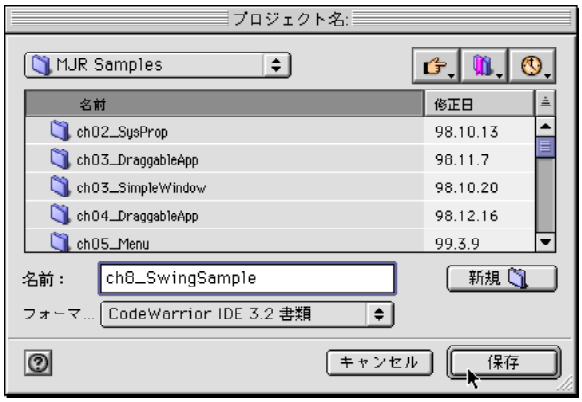

図 8-8　プロジェクトのファイル名とフォルダ名を指定する

こうして作られたプロジェクトのフォルダ（この例では、ch8_SwingSample）には、

プロジェクトのファイルと、アプリケーションのひな形となる TrivialApplication.java のファイルが作られているはずです。

まず、作られたプロジェクトのフォルダ内に Swing の本体である Swingall.jar ないしは Swing.jar ファイルをコピーしておきます。

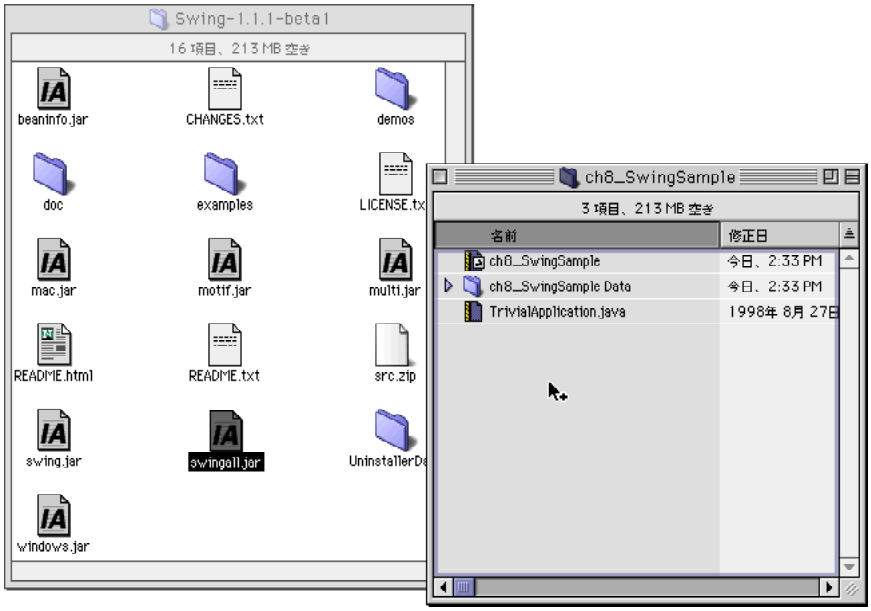

図 8-9　Swingall.jar をプロジェクトのフォルダにコピーする

また、MRJ の基本機能だけを使うのであれば、MRJ SDK の MRJToolkit フォルダにある MRJToolkitStub.zip もプロジェクトのフォルダにコピーしておくと便利でしょう。

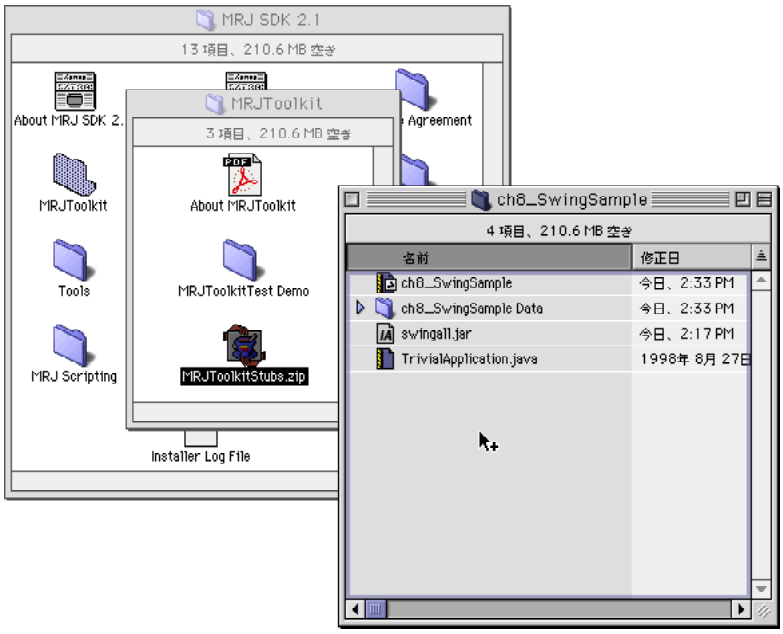

図 8-10　MRJToolkitStub.zip もコピーしておく

そして、Swingall.jar、MRJToolkitStub.zip を、プロジェクトのウインドウにドラッグ&ドロップして、プロジェクトに追加しておきます。

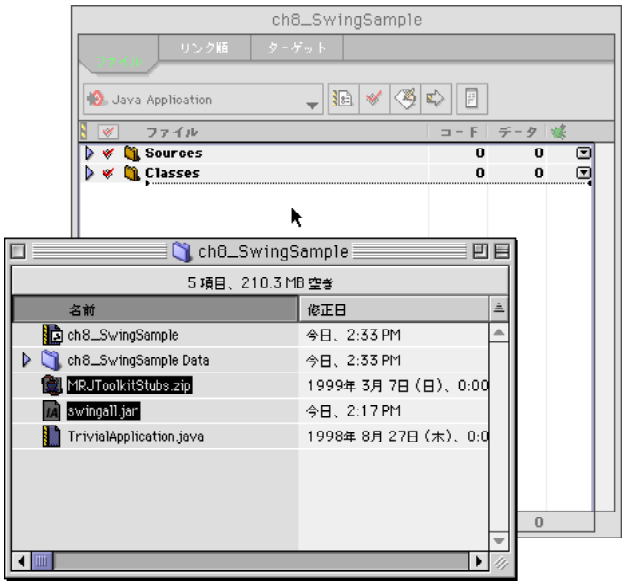

図 8-11　フォルダにコピーしたファイルをプロジェクトに登録する

ただし、この MRJToolkitStub.zip では、MRJ のすべての機能は利用できません。もし、MRJToolkit の機能を超えたプログラムを作成するには、システムフォルダの「機

能拡張」フォルダにある、MRJLibraries フォルダの MRJClasses フォルダにある MRJClasses.zip をプロジェクトに登録しておきます。Finder からドラッグ&ドロップしてもかまいませんが、「プロジェクト」メニューから「ファイルを追加」を選択して、機能拡張フォルダ内の MRJClasses.zip を選択して登録しても良いでしょう。

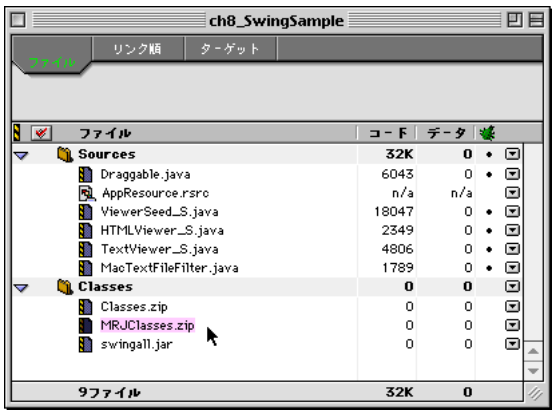

図 8-12　プログラムによっては、MRJClasses.zip をプロジェクトに登録する

なお、この章のサンプルプログラムは、MRJClasses.zip をプロジェクトに登録しておく必要があります。

8-2　Swing の機能を使う

Mac OS 上で Swing の機能を使った場合に留意すべき点などをまとめておきましょう。Swing は非常に広大なライブラリで、すべてのクラスについてチェックをしたわけではなく、サンプルプログラムの範囲内でのチェックとなりますが、ウインドウやメニューなどの基本的なことはチェックしてあります。

ルック&フィールの使い方

ボタンのデザインやあるいはメニューのデザインなど、GUI 部品の見かけをルック&フィールとして選択することができるのが Swing の大きな特徴です。Swing に最初からいくつかのルック&フィールが含まれていますが、ルック&フィール自体を Java で開発することができるので、場合によっては独自のものを作るということも可能です。実際、Mac OS 向けに別のルック&フィールがリリースされています。

まず、基本セットで利用できるルック&フィールを見てみましょう。swing.jar ライブラリに含まれている Metal という名前のルック&フィールは、Swing のデフォルトのルック&フィールと言える存在で、「特に指定しなければ」（ただし、この表現はやや間違いがあります）このルック&フィールを利用すると言えば良いでしょうか。文字通り金属的なイメージはしなくもないですが、ともかく、OS に関係なく利用することを意図した基本ルック&フィールのようです。

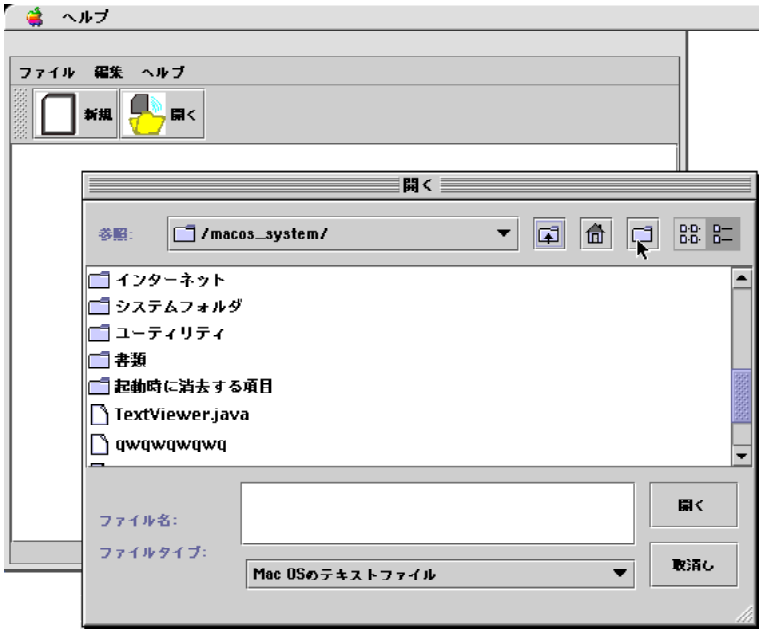

図 8-13　Metal のルック&フィールを選択したときの様子

ルック&フィールを設定する機能は、Swing のライブラリの中にある UIManager というクラスを利用します。このクラスにあるスタティックメソッド setLookAndFeel を使って、ルック&フィールの構築に使用するクラス名を文字列で指定します。

```
import javax.swing.*;
    :
UIManager.setLookAndFeel("javax.swing.plaf.metal.MetalLookAndFeel");
```

引数にある、javax.swing.plaf.metal.MetalLookAndFeel というのは、Metal ルック&フィールの処理を行うクラス名の完全な記述です。この MetalLookAndFeel というクラスは、swing.jar あるいは swingall.jar に含まれています。なお、setLookAndFeel メソッドは、例外の処理を組み込まなければなりません。

ライブラリとして、swingall.jar、あるいは swing.jar と windows.jar を使っていれば、Windows 風のルック&フィールにすることもできます。たとえば、ファイル一覧ではフォルダのアイコンが黄色だったり、ツールバーやダイアログボックスの色合いなどが、Windows 風になります。

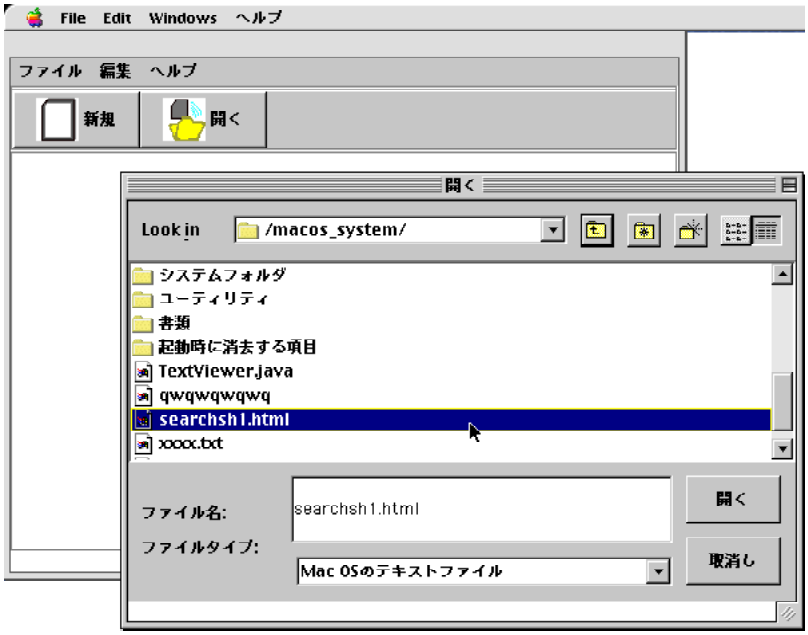

図 8-14　Windows **のルック&フィールで表示した**

Windows のルック&フィールに設定するには、やはり、setLoolAndFeel メソッドを使います。たとえば、次のような 1 つの命令があれば、それで Swing 全体のルック&フィールが Windows 風に設定されます。

```
import javax.swing.*;
    :
UIManager.setLookAndFeel(" com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
```

Windows のルック&フィールを提供するクラスは、com.sun.java.swing.plaf.windows.WindowsLookAndFeel となっており、階層が Metal と大きく異なっています。これは、Swing の Ver.1.0 時代に、すべてのクラスが、com.sun.java.swing 以下の階層で定義されていたなごりでしょう。Ver.1.1 から Javax の階層に変わったのですが、ソース中にクラス名の全階層を記述しているためにバイトコードにその文字列が含まれることになります。そこで、過去との互換性のために、ルック&フィールのクラスはそのまま com.sun…になっているのだと思われます。

同様に、Motif 風のルック&フィールを提供するクラスもあります。これは、swingall.jar を使っているか、swing.jar と motif.jar を使っている場合に利用できます。setLookAndFeel の引数に、com.sun.java.swing.plaf.motif.MotifLookAndFeel という文字列を指定すると、Motif 風のルック&フィールになります。

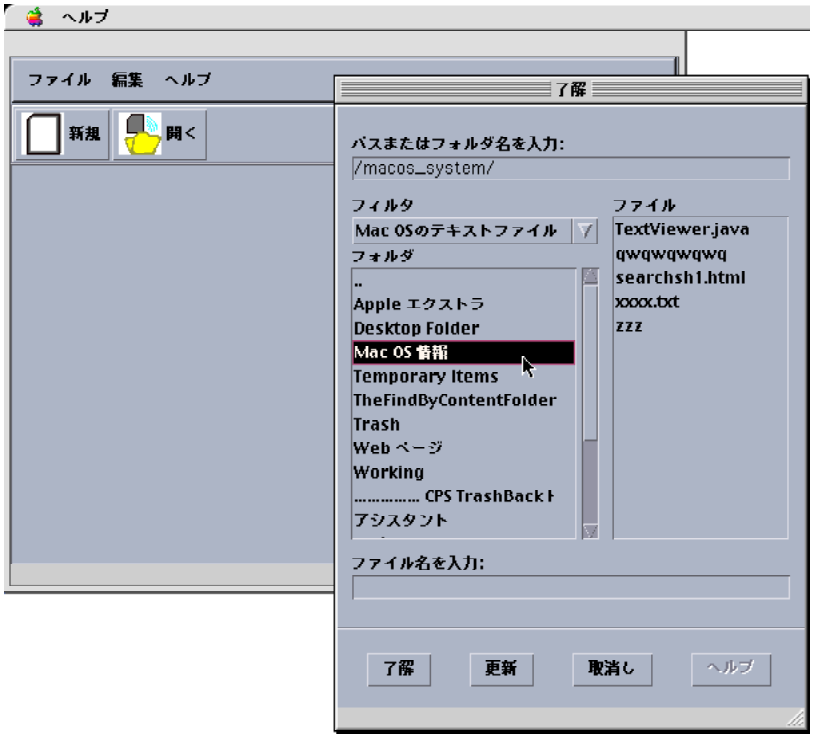

図 8-15　Motif **風のルック&フィールに設定した**

♦Mac OS 風のルック&フィールを使う

Swing には、Mac OS 風のルック&フィールも用意されています。ただし、このルック&フィールを実現するクラスは、swingall.jar には含まれていません。そこで、swingall.jar あるいは swing.jar のいずれを使う場合にでも、mac.jar はプロジェクトに含めなければなりません。mac.jar ファイルは、Swing によってインストールされる一連のファイルの中に含まれているはずです。

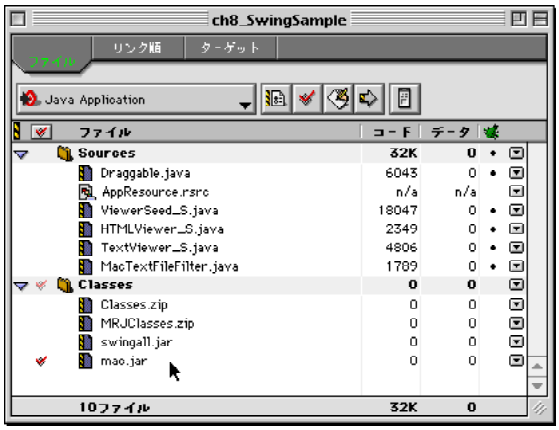

図 8-16　Mac OS **のルック&フィールを使うには、**mac.jar **をプロジェクトに含める**

プロジェクトに、mac.jar が含まれた状態で、次のように、setLookAndFeel メソッドを使うと、ルック&フィールは Mac OS 風になります。

```
import javax.swing.*;
    :
UIManager.setLookAndFeel("com.sun.java.swing.plaf.mac.MacLookAndFeel");
```

図 8-17　Mac OS のルック&フィールで表示した

Mac OS のルック&フィールを使った時には、そのときのアピアランスの設定も反映されます。この図は、ラージシステムフォントに Osaka Bold を指定したときのものなので、メニューが太字になっています。ただし、ウインドウに登録したメニューは、メニューバーではなく、ウインドウのタイトルバーのすぐ下に表示されており、その意味では Windows 的です。メニューについての問題は、別のところで検討しましょう。

また、図ではファイル選択ダイアログボックスが表示されています。これは、システムのファイル選択ダイアログボックスというよりも、Swing でカスタマイズ可能なファイル選択ダイアログボックスの機能があり、それを利用して Mac OS 風にしたという感じのものになっています。どうせ真似るなら、Navigation Service のダイアログボックスを真似てほしいとも感じるところです。

♦システム標準のルック&フィール

使用するルック&フィールを指定しない場合には、自動的に Metal が使用されます。

つまり、Metal がルック&フィールのデフォルトになっています。

一方、現在のシステムのルック&フィールを得る機能が Swing には用意されています。UIManager クラスにある getSystemLookAndFeelClassName メソッド（スタティック）を利用すると、現在のシステムに適したルック&フィールのクラス名を文字列で得られます。たとえば、次のような命令があれば、コンソールにシステムのルック&フィールのクラス名をコンソールに出力します。

```
import javax.swing.*;
    :
System.out.println("System Look & Feel: "+ UIManager.getSystemLookAndFeelClassName());
```

Mac OS 上で実行すると、getSystemLookAndFeelClassName メソッドの戻り値は、com.sun.java.swing.plaf.mac.MacLookAndFeel つまり Mac OS のルック&フィールになります。

注意したいのは、getSystemLookAndFeelClassName メソッドの戻り値は設定されている値ではなく、あくまで Swing によって推薦されるクラスだというところです。これで得られたクラスでルック&フィールを構築するには、setLookAndFeel クラスで改めて設定をしてやらなければなりません。逆にそうしたプロセスを組み込めば、実際に稼動している OS に近いルック&フィールを自動的に選ぶということも可能になります。

ところが、Mac OS のルック&フィールのクラスは、swing.jar はもちろん swingall.jar にも含まれていません。getSystemLookAndFeelClassName メソッドが、Mac OS のルック&フィールのクラスを戻したからと言って、必ずしもそのクラスが利用可能な状態になっているとは限らないのです。つまり、mac.jar をプロジェクトに必ず含めておかないといけないということになります。

Swing でのウインドウ

Swing のウィンドウに関する機能は、AWT のウインドウである Frame クラスと基本的にはほとんど同じように使えます。むしろ、共通に使えるメソッドが多いので、Swing だからと言って異なる点はほとんどありません。たとえば、表示や非表示はもちろん、AWT と同様のレイアウト機能を利用することができ、また、メニューなどの登録もできるので、基本的には AWT と同じように利用すればそれで済みます。

javax.swing.JFrame、つまり JFrame クラスが定義されており、たとえば、このクラ

スを継承することによって、Swing に対応したウインドウを作成できます。ウインドウは、その時に指定されたルック&フィールに従った見かけで表示されます。

♦Frame と JFrame の違い

Frame と JFrame の違いの 1 つは、コンポーネントの追加処理にあります。Frame 自体が Container のサブクラスであったため、ボタンなどのコンポーネントを Frame に対して add メソッドで追加できました。つまり、ウインドウである Frame に直接追加できたわけです。

一方、JFrame は直接の追加はできません。JFrame にある getContentPane メソッドを使って、ウインドウの中のコンポーネント追加可能な場所を取得し、そこに対して add メソッドでコンポーネント類を追加するようにします。プログラムは、AWT では直接 add していたところを、getContentPane メソッドを通じて得られたオブジェクトに対して add するように、一段階プロセスが増えるだけです。getContentPane の戻り値の型は AWT の Container です。

♦ウインドウのクローズは自動的にできる

Frame で作ったウインドウは、そのままではクローズボックスをクリックしてもウインドウは閉じられませんでした。そこで、イベントリスナを組み込んでクローズボタンをクリックしたことをクラスに伝えられるようにし、そのイベントを受けてウインドウをクローズしたり、あるいはアプリケーションをクローズするようなプログラムを組む必要がありました。

しかし、Swing の JFrame では、そうしたイベント処理は組み込む必要はありません。クローズボックスのクリックにより、自動的にウインドウは閉じられます。もちろん、閉じるときに何らかの処理を行うのであれば、イベントを取り扱うようにする必要はありますが、単に閉じるだけでいいのであれば、何の処理も組み込む必要はなくなったわけです。

Swing でのメニュー

メニューについては、やはり AWT に近い形式のクラスが Swing でも定義されていますが、いろいろな意味で違ってきます。また、Mac OS で動かすアプリケーションとなると、さらに考慮すべきことが出てきます。メニューについては Swing 独自の事情を認識しておかなければなりません。

まず、Swing のメニューを利用するための基本的なクラスを表にまとめておきまし

た。基本的には、AWT のクラス名に J が頭に付くと考えれば良いでしょう。

表 8-2　メニューで利用するクラス

構成要素	AWT のクラス	Swing のクラス
メニューバー	MenuBar	JMenuBar
メニュー	Menu	JMenu
メニュー項目	MenuItem	JMenuItem
ショートカット	MenuShortcut	(KeyStroke)
選択処理	ActionListener	ActionListener

メニューバーやメニュー、メニュー項目を生成して、それらを必要な要素に登録すれば、ウインドウのメニューは作成されます。メニューバーをウインドウに登録するには、JFrame で定義された setJMenuBar メソッドを利用すれば良いでしょう。

ちなみに、AWT の機能で作られたメニュー処理プログラムにおいて、Menu を JMenu に一括置換するだけで、基本的なところは全部置き換わり、Swing 対応のメニュー処理プログラムになります。ただし、違いのある部分のプログラム修正は必要になりますが、まずは一括置換をするというのが第一歩になるでしょう。

♦メニューを Mac OS のメニューバーに表示する

Swing のメニューは、基本的には必ずウインドウのタイトルバーの下に、ウインドウごとに表示されます。つまり、Windows などの GUI の形式なので、Mac OS のようなメニューバーにメニューを出すような形式にはなっていません。これを、Mac OS 的にする方法はなくはありません。

1 つは、Swing のメニューを使わないことです。JFrame を使っていても、AWT のメニュー処理でメニューを構築すると、MRJ 環境下では、メニューバーを使ったメニューが表示されます。その場合、メニューバーのウインドウへの登録は、JFrame に対して setMenuBar を使います。JFrame は Frame を継承しており、AWT のメニューに関連する機能は、Frame で定義されたメソッドを使うことになります。ただし、Swing のメニュー機能は非常に豊富で、メニューに文字やショートカットだけでなく、アイコンなどのグラフィックスやさらにはコントロールのようなものまで配置できます。そうした豊富な機能を望むのであれば、Swing を使い、ウインドウ内に表示されるメニューを使わざるを得ないでしょう。

もう 1 つの方法は、Sun Microsystems ではないところで開発された Mac OS L&F という独自のルック&フィールを使うことです。このルック&フィールを使えば、Swing のメニュー関連の機能を使っても、Mac OS のメニューバーにメニューが表示されま

す。ただし、Mac OS のメニュー機能に依存する面もあるので、やはり使えない処理も出てきて、それなりに制約があります。このルック&フィールについては、別にまとめておきましょう。

♦メニュー項目の生成とショートカットやアクセラレーターキー

メニューを組み立てる基本は、AWT と同様です。ただし、ショートカットキーの扱いは大きく違います。AWT のメニューの場合、1 つの方法として MenuItem のインスタンスを生成するときに、引数に MenuShortcut クラスのインスタンスを指定していました。それによってショートカットキーを割り当てるわけです。

一方、Swing では、用語の使い方が異なります。Windows ではメニューを選択するのに、たとえば Alt キーを押して、F キーを押して、O キーを押すと、「ファイル」メニューの「開く」を選択するような操作体系があります。それに利用するのを「キーボードニーモニック」と呼んでいます。一方、Ctrl+X や Command+X のように、キーボード処理一発で何かメニュー項目を呼び出す機能は「アクセラレーターキー」と呼んでいます。

話がややこしくなりますが、Windows では、Ctrl+X などのキー操作はショートカットと呼んでおり、Alt→F→O のような順序での F や O キーの事を、「アクセスキー」と呼んでいます。このあたりは用語が混乱しますが、以後は Swing 用語で説明することにします。

たとえば、メニューバーを作成し、メニューを作成し、さらにメニュー項目を作成する部分を抜粋すると次のようになります。

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

    :

JMenuItem newItem

JMenuBar mb = new JMenuBar();      //メニューバーを用意する

/*「ファイル」メニューの作成*/
JMenu fileJMenu = new JMenu("ファイル", true); //「ファイル」メニューを作成する

newItem = new JMenuItem("新規", KeyEvent.VK_N);    //「新規」項目を作成
newItem.setAccelerator(
          KeyStroke.getKeyStroke(KeyEvent.VK_N, ActionEvent. META_MASK ));
                                                  //ショートカットは N
fileJMenu.add(newItem);           //「ファイル」メニューに追加する
NewItemListener newListener = new NewItemListener();
```

```
                    //「新規」を選択したときにイベントを受け付けるクラスを新たに生成
newItem.addActionListener(newListener);
                    //「新規」を選んだときにイベントが発生するように設定
    :
    :

/*「ファイル」メニューの「新規」を選択、あるいは「新規」ボタンをクリックしたときの処理*/
class NewItemListener implements ActionListener      {
                public void actionPerformed(ActionEvent ev)  {
        newDocumentByViewer();          //新規ドキュメントを用意する
    }
}
```

まず、JMenuBar を生成してメニューバーを作成します。さらに JMenu を生成してメニューを作成しますが、1 つ目の引数はメニュー項目名、2 つ目の引数はティアオフメニューかどうかを示すのですが、現在はまだ機能は組み込まれていません。

そして、JMenuItem が実際のメニュー項目に相当します。生成するときのコンストラクタで 2 つの引数を取っていますが、1 つ目がメニュー項目名、2 つ目が int 型ならニーモニックキーを指定します。ちなみに、2 つ目が javax.swing.ImageIcon 型のアイコン画像なら、アイコン付きのメニュー項目を生成できます。

ニーモニックキーがどのキーかを指定するためには、java.awt.event.KeyEvent クラスに定義されたスタティック変数を使用します。このクラスには、VK_で始まるキーコードを記述した変数が定義されているので、それを利用して、キーを指定します。キーと変数の対応は、API のドキュメントを参照してください。たとえば、VK_N は、「N」のキーを示しています。なお、JMenu にニーモニックキーを指定する場合には、setMnemonic メソッドを使用し、引数に KeyEvent クラスのスタティック変数を指定します。

このニーモニックキーを使ったメニュー選択は、Metal のルック&フィールのときは、F10 キーを押します。そうすれば、この章のサンプルプログラムだと「ファイル」メニューが開きます。そこで、たとえば「N」キーを押すと、「新規」が選択されるという具合です。各ルック&フィールでのキー操作については、setLookAndFeel で指定するクラスの API ドキュメント中にリンクがあり、そこをクリックすると一覧表で参照できるようになっています。いきなりメニューが開くあたりは、意図的かどうか、Windows との違いを出しています。ただ、ニーモニックキーについては完全に日本語化されていない面があります。英語だと、New という項目のニーモニックキーが N だとしたら、New という単語の N の部分に下線が引かれます。しかしながら、「新規」というメニュー項目に N をニーモニックキーと指定しても、N がニーモニックであるという情報はメニュー項目には表示されません。Windows では、「新規(N)」のような

記述をしてニーモニックに相当するアクセスキーを指定していますが、日本語だとそうしたメニュー名を文字列として与えないといけなくなります。

そして、アクセラレーターキーは、生成した JMenuItem のインスタンスに対して、setAccelerator メソッドを使用して設定します。このメソッドの引数には、javax.swing.KeyStroke を指定しますが、このクラスに KeyStroke を生成するスタティックなメソッド getKeyStroke を使用します。このメソッドはオーバーロードされていて、引数に対して複数のバリエーションがあります。しかしながら、メニューのアクセラレーターキーは一般には、文字キーと修飾キーの組み合わせになるでしょう。1 つ目に文字キーを、2 つ目に修飾キーを指定します。文字キーはニーモニックキーと同様、java.awt.event.KeyEvent のスタティック変数を指定します。そして、修飾キーは、java.awt.Event クラスのスタティック変数を利用します。Ctrl キーは CTRL_MASK、Shift キーは SHIFT_MASK になります。Command キーは META_MASK になります。option キーに対応する機能はないようです（ALT_MASK で ALT キーを指定してもだめでした）。複数の修飾キーを指定する場合には各変数を加えればよいでしょう。このあたり、まじめにマルチプラットフォームソフトを作る場合には、OS によってアクセラレーターキーを異なるものに設定することになるでしょう。

こうしてアクセラレーターキーの指定を行えば、そのキー操作により自動的にメニュー項目に指定したイベント処理が行われます。つまり、キー操作を受け付けるイベント操作を定義する必要はありません。

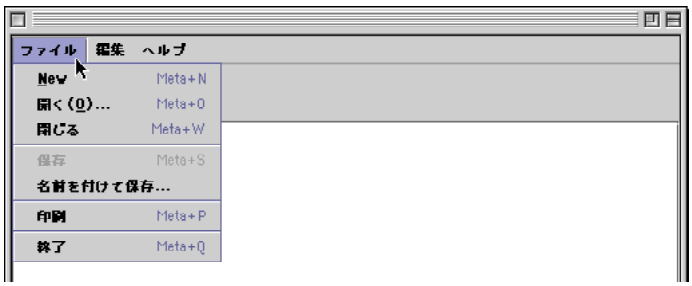

図 8-18　作成したメニューの一例

作成したメニューを見てみると、まず、Command キーなのに、メニュー上のアクセラレーターキー表示が、Meta となっていることです。これは、Mac OS の標準のルック&フィールを使っていても同様です。Command キー独特のマークが出てこないと、Mac OS の利用者は不満に思うところでしょう。

また、別の問題があります。テキスト編集可能な JTextArea をウインドウ内に配置

してある場合、たとえば、カットしようとして Command+X を押すと、「x」という文字が入力されてしまうのです。Command キーを押した状態でキーボードを押すと、どれもキー入力されてしまうので、アクセラレーターキーなら入力されないようにするような処理がどこかで欠けているのではないかと思います。これをプログラムで回避するのはかなり複雑そうですし、第三者が作った Mac OS 向けのルック&フィールではこの点は解決されています。この点は、Swing 自体で修正されるのを待った方が良いと思われます。

ヘルプメニューに関する処理が AWT のメニューに組み込まれています。Swing にもメソッドは用意されているのですが、まだ機能が組み込まれていないという風に API ドキュメントで解説されています。ヘルプメニューへの追加は、現在のバージョンでの Swing では行わないようにしておく必要があります。

♦MacOS L&F を使う

Luca Lutterotti 氏が、Swing に付属する Mac OS 向けのルック&フィールをもとに改良したものを配付しています。Swing オリジナルのものを Mac L&F と呼び、Luca Lutterotti 氏のものは区別するために、Mac OS L&F と同氏は呼んでいます。

Mac OS L&F の大きな特徴は、JMenuBar で作成したメニューは、ウインドウのタイトルバーの下ではなく、Mac OS のデスクトップとしてのメニューバーにきちんと表示されることです。また、Command キーのショートカットは、きちんと“クローバーマーク”でメニューに表示されます。単にルック&フィールを切り替えるだけで、より Mac OS に適合できるのです。こうしたルック&フィールを開発していることは賞賛できることですが、このようなプラグイン的な機構を持っている Swing の柔軟性も注目できるところでしょう。

この Mac OS L&F は、Luca Lutterotti 氏の Web ページである「Java On Mac」（http://www.ing.unitn.it/~luttero/javaonMac/）から入手できます。いくつかのパターンがありますが、ルック&フィールを実現するクラスを含む、macos.jar というファイルが利用できれば良いでしょう。

ただし、この Mac OS L&F を使うためには、パッチの当たった Swing のライブラリである swingall.jar を利用しなければなりません。実際、Swing 1.1.1β1 の swingall.jar を利用するようにしていると、正しく機能しません。

プロジェクトで Mac OS L&F を利用するには、たとえば、次のように作業します。まず、Java On Mac のページからダウンロードしたファイルを解凍します。ここでは「SwingSetMacOS」というフォルダに展開したとします。その中にある swingall.jar と

macos.jar の 2 つのファイルを、作成中のアプリケーションのフォルダ（ch08_SwingSample）にコピーします。コピー先にオリジナルの swingall.jar がある場合には、たとえばフォルダを作っておくなどして隠しておけばよいでしょう。

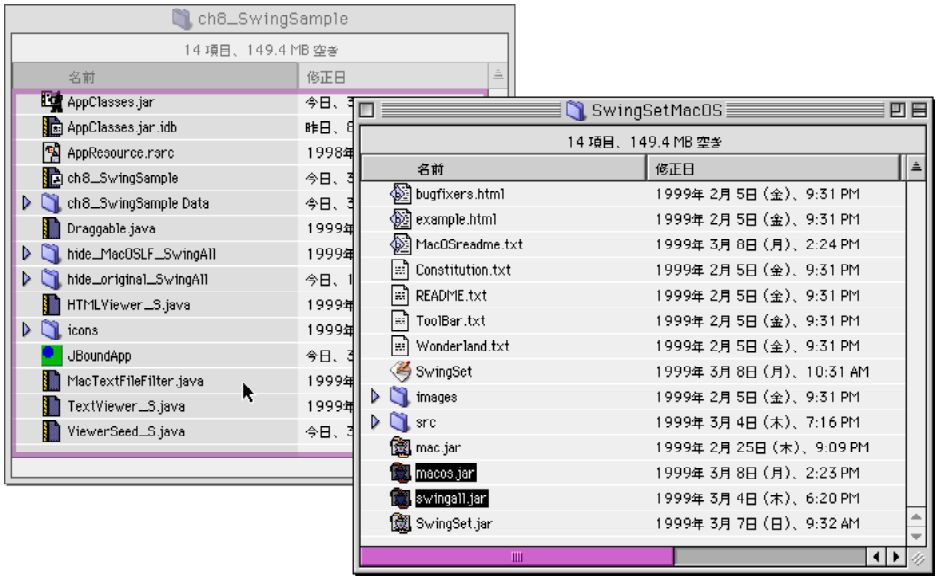

図 8-19　macos.jar、swingall.jar をコピーする

そして、それらコピーした 2 つのファイルをプロジェクトに登録します。念のため、いったん swingall.jar をプロジェクトから削除し、その上で、swingall.jar と macos.jar をプロジェクトのウインドウにドラッグ&ドロップして登録すればよいでしょう。

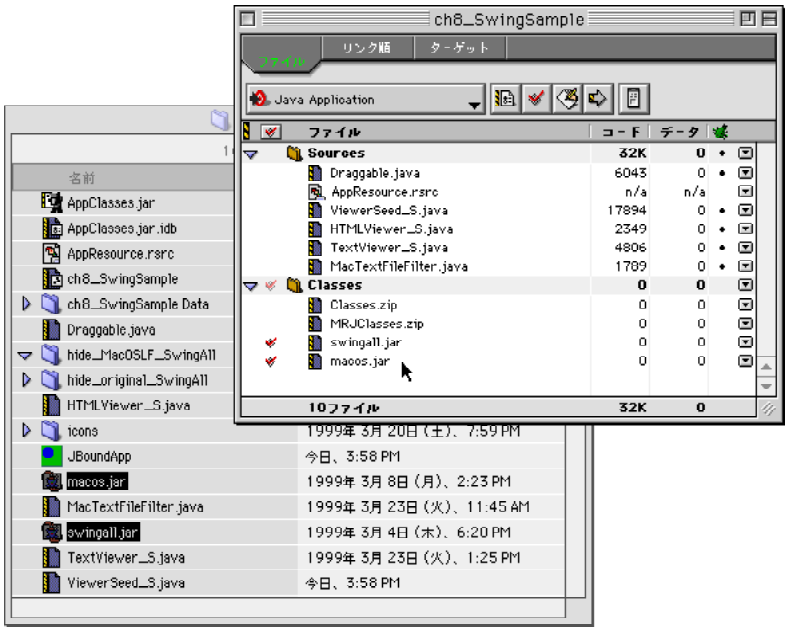

図 8-20　macos.jar、swingall.jar **をプロジェクトに登録する**

なお、swingall.jar を、Mac L&F のものとオリジナルの物で切り替えて使うには、それぞれフォルダに隠ししたり出したりすればいいのですが、その都度プロジェクトへ登録しなおす必要はないようです。CodeWarrior は、アクセスパスで認識するフォルダにある swingall.jar ファイルを取り込むので、エイリアスなどを使用しているわけではなく、あくまでファイル名で認識しているようです。従って、必ずしもプロジェクトへの登録しなおしは必要はないというわけです。

Mac OS L&F を使うには、以下のような命令を実行します。これによって、システム全体のルック&フィールが Mac OS L&F になります。Mac OS L&F を利用するためのクラスは以下のプログラム中の引数文字列を参照してください。実際にはさらに例外処理が必要になります。

```
UIManager.setLookAndFeel("com.sun.java.swing.plaf.macos.MacOSLookAndFeel");
```

こうして Mac OS L&F を使って表示させたアプリケーションを見てみると、メニューはきちんと Mac OS のメニューバーに表示されています。また、アクセラレーターキーの表示も、Mac OS 的にクローバーマークとアルファベットで表示されています。

図 8-21　Mac OS L&F **で表示した**

ただし、メニューの区切り線が表示されません。JMenu にある addSeparator メソッドが機能していないようです。また、システムのヘルプメニューに追加する機能もドキュメントでは説明されていますが、英語の Help という名称のメニューでないといけないようで、日本語システムのヘルプメニューへの追加もできないようです。

このように、まだ不完全だとは言え、Swing をより Mac OS 的にするルック&フィールとして、Mac OS L&F は注目しておく必要があるでしょう。

ツールバー

Swing にはツールバーのためのクラスが用意されている点が、AWT とは大きな違いです。ツールバーを管理する JToolBar クラスがあり、ツールバー内にはさまざまなコンポーネントを配置できます。一般には、JButton クラスで作るボタンを配置するのが手軽でしょう。JButton をインスタンス化するときに、引数を 2 つ指定し、1 つ目にボタンに表示する文字、2 つ目にアイコンを指定すれば、アイコンと文字の含まれたボタンが作成されます。アイコンは、IconImage クラスを利用しますが、これもコンストラクタの引数に GIF ファイルへのパスを指定すれば、その GIF ファイルをアイコンのデザインとしたアイコンを生成できます。また、JButton には、setToolTipText メソッドを利用して、ツールチップ、つまりマウスポインタをボタンの上に移動させたときに、ボックスが出てきてそのボタンの機能説明などを行うことができます。これも、メソッド 1 つだけで実現できます。

ボタンをクリックしたときに何か処理をする場合には、メニューとまったく同様に、アクションイベント処理を組み込みます。ActionListener をインプリメントしたクラスを呼び出すような仕組みを利用します。

最終的には、JToolBar を、JFrame の中に配置します。JFrame のコンテナを getContentPane メソッドで得て、それに add メソッドでツールバーを追加します。このあたりは、他のコンポーネントと同様に扱います。

以下はサンプルプログラムの一部です。JButton の引数にあるように、「icons/new.gif」のようなパスを指定しますが、これにより、アプリケーションのあるフォルダの icons フォルダにある new.gif という画像ファイルをアイコンに設定することができます。相対パスの場合には、アプリケーションの存在するフォルダが基点になるようです。JFrame のコンテナでは、BorderLayout の機能を利用しているため、NORTH の位置にツールバーを配置すると、ちょうどタイトルバーやメニューバーが表示されるすぐ下のよくある位置にツールバーが配置されることになります。

```
import javax.swing.*;
    :
JToolBar tb = new JToolBar();   //ツールバーを新しく作成する

JButton newButton = new JButton("新規", new ImageIcon("icons/new.gif"));
        //ボタンを生成する。ボタン名は「新規」で指定した GIF ファイルのアイコンをつける
newButton.setToolTipText("新しくウインドウを開く"); //チップヒントの文句
tb.add(newButton);          //ボタンをツールバーに登録する
NewItemListener newListener = new NewItemListener();
        //「新規」をクリックしたときにイベントを受け付けるクラスを新たに生成
newButton.addActionListener(newListener);
```

```
        //「新規」をクリックしたときにイベントが発生するように設定
    :
this.getContentPane().add(tb, BorderLayout.NORTH);
    //ツールバーをウインドウに登録する。BorderLayout の機能を利用し、
    //      ウインドウの上端に配置する
```

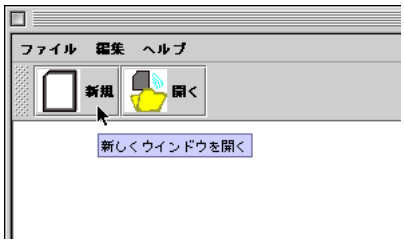

図 8-22　ウインドウに追加したツールバー。ツールチップを表示させた

なお、JToolBar は、特に何もプログラムを組まなくても、ドラッグして移動することができます。最初の状態のツールバーの左端は、ざらざらしたデザインになっていますが、ここをドラッグすることでフローティングツールバーになります。また、ツールバーの左端をドラッグして、ウインドウの上下左右の端に移動させると、その位置でウインドウにツールバーがドッキングします。なお、フローティングウインドウにした場合、常に手前に表示されるパレット形式にはならず、ウインドウの背後に隠れてしまうこともあるので、そのあたりの操作では注意が必要になります。

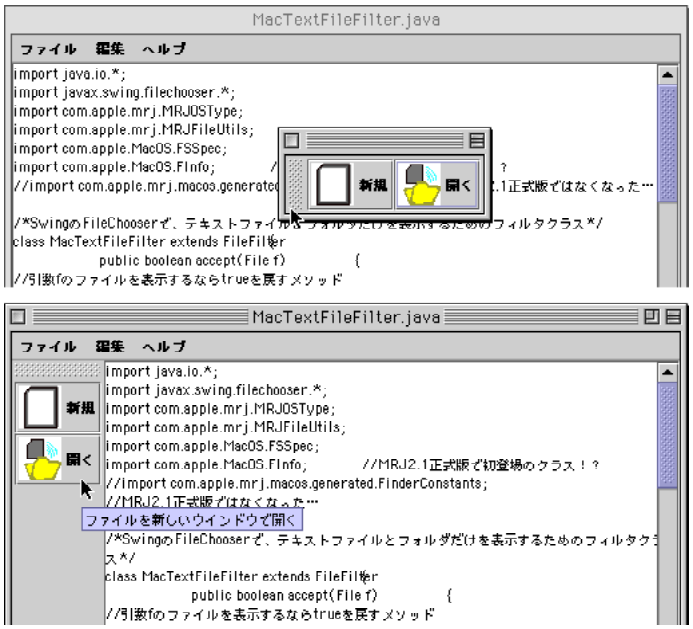

図 8-23　フローティングやあるいは左に配置したツールバー

Swing のコンポーネント

この章のサンプルプログラムでは、テキスト表示の JTextArea と HTML 表示の JEditorPane を使ったビューア的なアプリケーションを組んでみました。それらを利用した範囲内にはなりますが、Swing でのコンポーネントの様子をまとめておきましょう。

♦テキストを表示する JTextArea

テキストを開いて表示する場合は、AWT だと TextArea を使っていましたが、Swing ではその上位互換コンポーネントとでも言える JTextArea があります。こうしたコンポーネントは、AWT の場合は単にウインドウやアプレットに追加すればよかったのですが、Swing の JFrame では、まず、getContentPane メソッドを使って、コンポーネントを追加できるコンテナを得ます。さらにテキスト領域をスクロールさせたい場合には、スクロール領域の JScrollPane クラスのインスタンスを作成する必要があります。つまり、テキスト領域の JTexArea を生成し、それを含む JScrollPane を生成する必要があります。その JScrollPane を、getContentPane で得られた JFrame のコンテナに add メソッドで追加します。以上の手順が、スクロール可能なコンポーネントの基本的な追加手順です。つまり、スクロールとテキスト表示が TextArea ではまとまっていたのが、Swing では別々のコンポーネントで処理をするということになります。

JTextArea は、TextArea と違って、ウインドウ幅で折り返すような表示も可能になっています。setLineWrap(true)で折り返しするようになり、setWrapStyleWord メソッドで、ワード単位の折り返しをするかどうかの指定もできます。ただし、折り返しをすると処理速度は目に見えて落ちるくらいになります。

以上のように、テキストデータを表示する JTextArea があって、その一部分をスクロールすることで見せる JScrollPane があります。たとえば、BorderLayout がレイアウト機能として利用されている場合には、JScrollPane をその Center 領域に追加します。

そして、これらのいくつかのコンポーネントの大きさを調整しなければなりません。ウインドウのサイズを変更したときの大きさ調整は、BorderLayout を利用していれば自動的になされます。レイアウト機能を使わない場合には、独自にサイズ調整が必要になるでしょう。プログラム上でサイズ調整をしなければならないのは、ウインドウを作成した最初の段階です。いろいろやってみた結論として、まず、最終的に JFrame の pack メソッドを使って最終調整をすることになるのですが、そこで望む大きさにウインドウがなるように、この場合だと JScrollPane のサイズを調節します。単にサイ

ズを調節するのではなく、setPreferredSize メソッドを使って、望ましいサイズ（Prefered Size）の設定を行います。pack メソッドは望ましいサイズに設定するメソッドです。それで、ウインドウに含まれる JScrollPane を設定します。そして、JTextArea については、自動的にサイズが設定されるようです。おそらく、pack メソッドを実行した段階で、ウインドウに含まれるコンポーネントの全体的なサイズ調整が行われるのではないかと思われます。

JTextArea に文字列を設定するには、TextArea と同様、append メソッドなどが使えますが、この章のサンプルプログラムのようにテキストファイル、一般にはストリームからの入力が分かっているのであれば、read メソッドで、ストリームから読み込んだテキストをテキスト領域に設定します。そのとき、Reader クラスを指定するので、ファイルから読み取るのであれば、FileReader クラスを生成して、それを指定すると、FileReader クラスが管理するテキストファイルの内容を JTextArea に取り込みます。同様に、JTextArea の中身を書き出すには、Writer クラスを指定して write メソッド 1 つでストリームに書き出します。ファイルに書き出すには、FileWriter を生成します。これらのメソッドは、Mac OS 上で実行した場合、Mac OS 向けの動作をします。read は改行コードの種類に関係なく読み取りを行うようです。write は、Mac OS では CR コードだけを書き出します。

JTextArea で非常に便利なのは、クリップボード処理がほとんど 1 つのメソッドでできることです。文字通り、cut、copy、paste のメソッドがあり、クリップボードのデータの取得などをプログラミングする必要はありません。また、これらのショートカットとして、Ctrl+X、C、V は自動的に働くようになっています。ただし、Command+X、C、V は、メニュー処理のショートカットとして呼び出さないといけません。メニューのところで指摘したように、メニューのショートカットとしては機能するものの、副作用として、X、C、V の文字まで入力されてしまいます。このあたりは、Swing 側の対応を待つのが良さそうです。

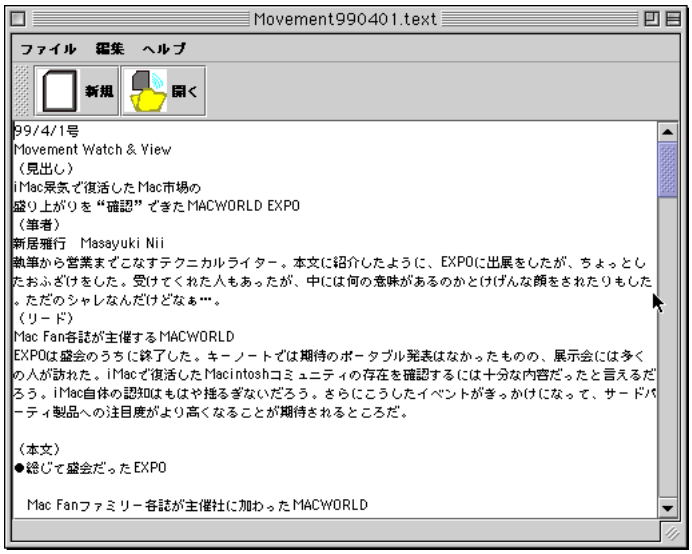

図 8-24　**テキストファイルの内容を表示した** JTextArea

♦HTML **エディタを作成できる** JEditorPane

Swing の JEditorPane クラスを使えば、HTML テキストの表示に加え、内容の編集もできます。ただし、文字の編集程度ならともかく、タグが絡むような編集作業は単に配置しただけではすべては行えないのですが、メニューやツールバーに機能を用意すれば、これで HTML エディタ並のワープロが簡単に作れるのではないかと期待してしまいます。JEditorPane を使うだけで、リンク先もテキストとして編集できます。リンク先をクリックしてジャンプするブラウザの機能にすることも可能で、リンクをクリックしたというイベント処理を組み込みます。この章のサンプルではそこまでの機能は含めていません。

JEditorPane の使い方は、JTextArea と基本的には同じです。JEditorPane のコンストラクタの引数に URL を指定すれば、その URL の中身を表示する JEditorPane が作成されるので、read による読み込みも場合によっては必要ありません。

以下の図は、GoLive CyberStudio3 で作った HTML ページのファイルを、JEditorPane で表示した例です。完全に HTML を表示しているのではなく、タグはタグとして表示する場合もあるようです。いずれにしても、HTML エディタを簡単に作れるというあたりに非常に可能性を感じるというところでしょうか。

図 8-25　CyberStudio **で作ったページを** JEditorPane **で表示した**

♦ファイル選択ダイアログボックス

Swing にはファイル選択のダイアログボックスも、javax.swing.FileChooser というクラスで提供されています。AWT とは別の機能となっていて、どちらかと言えば、Windows のファイル選択ダイアログボックスに近いデザインになっています。開くや保存の処理はもちろん、カスタマイズしたダイアログボックスも表示できます。前にも説明したように、Mac OS 向けのルック&フィールでは、このファイル選択ダイアログボックスは、従来システムの標準ダイアログのデザインを、さらに真似して作ったような使い勝手の悪いものになっています。Metal などのルック&フィールに比べても機能的に削られている面もあり、残念なところです。ダイアログボックスを Swing の機能で生成していることもあるのでしょうが、いずれにしても、Navigation Service の機能は使えません。一方、MJR 2.1 以降では、AWT の FileDialog クラスを使えば、Navigation Service 対応のダイアログボックスが表示されます。実用性を考慮すれば、現状では Swing のファイル選択ダイアログボックスは使わないで、AWT の機能を利用する方がベターな気がします。

また、Swing のファイル選択ダイアログボックスでは、フォルダの階層がどう見ても余分に付きます。フォルダ階層のドロップダウンリストを見れば、フォルダ階層にある最後に、フルパスの階層が付いています。ファイル一覧には、漢字など日本語は正しく表示されますが、スラッシュが%2f で表示されるなど、Finder で見えるファイル名とは必ず一致していません。つまり、Java のシステムとして都合のいいファイル名の文字列になっているというわけです。

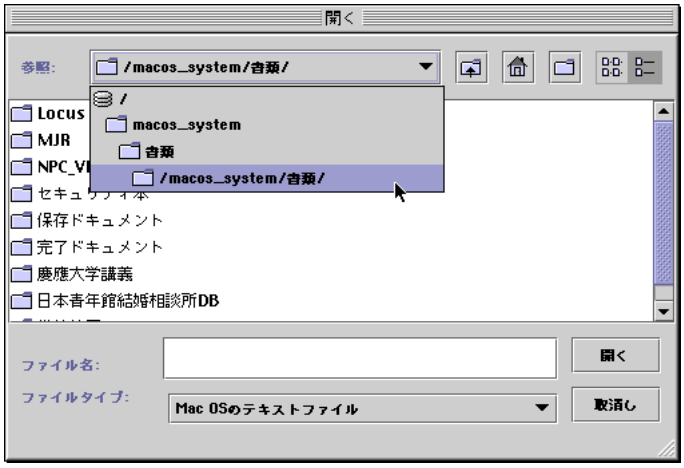

図 8-26　ファイル選択ダイアログボックスでの上位フォルダへの移動コントロール

Swing のファイル処理では、プラットフォームに依存するようなファイル処理も可能なように、javax.swing.filechooser.FileSystemView という抽象クラスが定義されています。実際には、このクラスを継承したきちんと働くクラスをインプリメントして置く必要があるのでしょうけれども、どうも完全には機能しません。このクラスには、ボリュームのルートへの参照を得られたり、隠しファイルかどうかの判断ができるメソッドが定義されているので、java.io.File クラスの機能に足りない面は補完できるはずなのです。

FileSytemView は、スタティックメソッドの getFileSystemView でインスタンスを取得します。ところが、isHiddenFile メソッドはすべて false を戻します。また、ボリュームの一覧を取ることもできますが、起動ボリュームかどうかの判断ができないため、Mac OS では使いづらい機能です。ドライブ番号が固定の Wintel マシン向けの機能のような気がします。

FileChooser では、フィルタ関数を定義してやることで、たとえばテキストファイルだけをダイアログボックスに表示するということもできるのですが、この章のサンプルでは、いろいろ制約が出てきてしまいました。その部分はどうしても JDirect の

機能に頼ることになってしまいます。

さらに、Swing のコンポーネントが戻すファイル名の文字列を、たとえば File などの生成で利用したときなどに、Mac OS のエラーを戻すときがあります。システムの隠しファイルなどで、実際に表示すると化け文字になるようなキャラクタが入っているときにこの現象が発生します。

♦Swing への雑感

機能豊富なコンポーネント群という意味では大いに期待できるのは言うまでもありません。実用的に利用できれば、アプリケーションは非常に作りやすくなります。

一方、現状の Swing が実用的かどうかは一概には言えないでしょう。むしろ、このような状態では、まだ実用的でないと判断される場面が出てくるのではないかと思います。1 つの理由は処理速度の遅さです。ファイル選択をネットワークの先のボリュームで行うと恐ろしく待たされます。Java の弱点だった処理の遅さも次第に解決されてきた矢先、Swing を使うとなると昔に戻されたような気がします。もちろん、近い将来はもっと高速なパソコンが一般的になるのだということを目論んでいるのであれば、必ずしもデメリットにはなりませんが、現状ではかなり高速な機種でないと実用的には使えないと思われます。

また、ファイル選択ダイアログボックスに見られるように、完成度が高くない面もあります。こうした部分を回避しながらプログラムを作るとなると、かなり大変になってきて、「手軽さ」というメリットが削がれると言えるでしょう。

Swing は非常に便利なコンポーネントですが、すべて Swing でそろえずに、場合によっては AWT のものを使うというのが、1 つの妥協点ではないかと思われます。

8-3 サンプルプログラム

Swing を利用したサンプルプログラムのソースを掲載します。テキストビューア、HTML ビューアの機能を持つようなアプリケーションを作成しました。また、BNDL リソースを与えて、ドラッグ&ドロップなどでもファイルを開くようにしています。

実行するための条件

サンプルアプリケーションは、5 つのソースファイルから構成されます。第 4 章の CodeWarrior でのアプリケーション作成方法に従って、プロジェクトを構築します。既定値からの違いは、まず、Mac OS Java Linker をポストリンカとして設定すること、MRJClasses.zip をプロジェクトに加えることは、まずはアプリケーションの基本です。また、BNDL リソースを含むファイルをプロジェクトに追加します。そして、ファイルタイプ rsrc に対して JarImporter を使うように設定を変更し、リソースファイルのインスペクタを開いて、リソースを生成ファイルにマージするようにしておきます。main メソッドがあるのは Draggable というクラスです。以上のような、アプリケーションとしての基本的な設計を行っておきます。

さらに、swingall.jar あるいは、swing.jar とルック&フィールのファイルなど、必要な Swing のコンポーネントをプロジェクトに追加しておきます。

サンプルプログラムは、第 5 章のプログラムをもとに Swing 対応しましたが、ビューアの種類が 2 種類になっているので、ビューアの元になるクラスと、実際のビューアのクラスに別れて、共通の処理を元になるクラスで記述するようにしています。

なお、サンプルプログラムのファイルはダウンロードできるようにしておき、一連のファイルにプロジェクトファイルは作成しておきますが、Swing のライブラリファイルは含めていません。Swing は独自に入手して、プロジェクトのフォルダに追加してください。

Draggable.java

このクラスに main メソッドを配置していますが、アプリケーションの基礎になる部分を定義しています。必須の AppleEvent である OpenDocument などに対応する処理

をここで組み込んでいます。また、ウインドウが何も表示されていないときにメニューが表示されるように、画面外にウインドウを表示して AWT の機能を使ってメニューを作成しています。

```
import com.apple.mrj.*;
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Draggable extends Frame        //Mac OS のアプリケーションの基本となるクラス。
                                            //      Frame を継承した
    implements      MRJOpenDocumentHandler,     //イベントハンドラのインプリメント
                    MRJPrintDocumentHandler,
                    MRJQuitHandler,
                    MRJAboutHandler
{
    static public void main(String arg[])      //起動時に実行されるメソッド
    {
        new Draggable();        //このクラスを新たに生成する
    }

    public Draggable()      //コンストラクタ
    {
        /*基本 AppleEvent が、このクラスに伝達されるように定義する*/
        MRJApplicationUtils.registerOpenDocumentHandler(this);      //イベントハンドラ
        MRJApplicationUtils.registerPrintDocumentHandler(this);     //への登録
        MRJApplicationUtils.registerQuitHandler(this);
        MRJApplicationUtils.registerAboutHandler(this);

        setupMenuBar(); //メニューバーの設定

        setBounds(-1000,-100,10,10);    //ウインドウの位置を画面の外側に出す。
            //      つまり非表示にしたいのだが、hide()とするとメニューまで消えるので、
            //      こういう方針にした
        show();     //ウインドウを表示するが、実際には画面には見えない
    }

    public void handleOpenFile(File filename)       //Open イベントが Finder からやってきたとき
    {
        System.out.println("Dragged "+filename.toString()); //コンソールへの出力
        ViewerSeed_S.openDocumentByViewer(filename);
                                            //開くファイルを TextViewer_S で表示する
    }

    public void handlePrintFile(File filename)      //Print イベントが Finder からやってきたとき
    {
        System.out.println("Print Event "+filename.toString());     //コンソールへの出力
        ViewerSeed_S newViewer = ViewerSeed_S.openDocumentByViewer(filename);
                                            //開くファイルを TextViewer_S で表示する
        newViewer.printDocumentByViewer();  //開いたビューアで印刷を行う
    }
```

```
//Quit イベントが Finder からやってきたとき、
//      あるいはアップルメニューの「終了」を選択したとき
public void handleQuit()
{
        System.out.println("Quit Event Received");     //コンソールへの出力
        System.exit(0);                                 //アプリケーションの終了
}

public void handleAbout()     //アップルメニューの About を選択したとき
{
        System.out.println("Select About Menu");       //コンソールへの出力
}

/*      TextViewer_S のウインドウが何も表示されていないときにもメニューを表示される
        ようにしたいそのような場合に必要なメニューに絞って表示するが、メニューを
        表示するにはそもそも Frame でなければならない。このクラスは Frame を拡張した
        が、そのウインドウは表示したくないので、コンストラクタで画面外に追いやった
*/
private void setupMenuBar()
{
        MenuItem newItem, openItem, closeItem, quitItem;

        MenuBar mb = new MenuBar();    //メニューバーを新たに生成

        Menu fileMenu = new Menu("ファイル", true);        //「ファイル」メニューを作成
        newItem = new MenuItem("新規", new MenuShortcut('N'));
                                        //「新規」の項目を生成し、ショートカットは N
        fileMenu.add(newItem);       //項目をメニューに追加する
        openItem = new MenuItem("開く...", new MenuShortcut('O'));
                                        //「開く」の項目を生成し、ショートカットは O
        fileMenu.add(openItem);      //項目をメニューに追加する
        fileMenu.addSeparator();     //区切り線を入れる
        quitItem = new MenuItem("終了", new MenuShortcut('Q'));
                                        //「終了」の項目を生成し、ショートカットは Q
        fileMenu.add(quitItem);      //項目をメニューに追加する
        mb.add(fileMenu);            //メニューをメニューバーに追加する

        NewItemListener newListener = new NewItemListener();
                                        //「新規」の処理クラスを生成する
        newItem.addActionListener(newListener);
                                        //メニュー項目を選択すると呼び出されるようにする

        OpenItemListener openListener = new OpenItemListener();
                                        //「開く」の処理クラスを生成する
        openItem.addActionListener(openListener);
                                        //メニュー項目を選択すると呼び出されるようにする

        QuitItemListener quitListener = new QuitItemListener();
                                        //「終了」の処理クラスを生成する
        quitItem.addActionListener(quitListener);
                                        //メニュー項目を選択すると呼び出されるようにする

        setMenuBar(mb);       //メニューバーをウインドウに登録する
```

```
    }

    class NewItemListener implements ActionListener  {
        public void actionPerformed(ActionEvent ev)  {      //「新規」で呼び出される
            ViewerSeed_S.newDocumentByViewer();      //ウインドウを新たに作成する
        }
    }

    class OpenItemListener implements ActionListener {
        public void actionPerformed(ActionEvent ev)  {      //「開く」で呼び出される
            ViewerSeed_S.openDocByViewerWithDialog(Draggable.this);
                //ダイアログを表示してファイルを指定し、そのファイルを開く
        }
    }

    class QuitItemListener implements ActionListener  {
        public void actionPerformed(ActionEvent ev)  {      //「終了」で呼び出される
            System.exit(0);  //アプリケーションを終了する
        }
    }
}
```

ViewerSeed_S.java

ビューアの元になる抽象クラスです。まず、ビューアはいずれもメニューを持っていたり、ツールバーを持っている点は共通です。ここでは、ビューアごとにメニューを変えるようなことも行っていないので、メニュー構築やツールバー構築を、このクラスで最初から行っています。また、メニューやボタンクリックなどのイベントの受付も内部クラスで行っています。ただし、実際の処理はビューアに使うコンポーネントに依存するので、abstract で定義したメソッドを呼び出すことにし、実際の処理はビューアごとに作成するようにしています。

また、インスタンス化するときに、使用するルック&フィールを設定しています。プログラム的には、単に設定するルック&フィールの命令をコメントからはずすようにしているだけですので、適当にコメントをつけたりはずしたりして、任意のルック&フィールを使用して下さい。

このクラスの中には、新しくビューアのウインドウを表示するのと、指定したファイルをビューアで開くメソッドを用意していますが、それらは static にしています。ファイルを開くメソッド内では、拡張子が html ないしは htm だと HTML ビューア、そうでないなら JTextArea のビューアを利用するようにしています。こうした機能は開いているビューアから呼び出されるだけでなく、OpenDocument イベントがあった

ときに Draggable クラスから呼び出されるなど、いろいろな利用パターンがあります。

そこで、static にして、汎用ルーチン的な定義にしてみました。

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import javax.swing.*;
import com.apple.mrj.*;

/*  ビューアのもとになるクラス。Swing 対応で、JFrame を継承している    */
abstract class ViewerSeed_S extends JFrame
{
    File openFile = null;           //開いているファイル
    Dimension initilaWindowSize = new Dimension(500,300);       //ウインドウの初期サイズ
    boolean isDirty = false;        //修正したかどうかを記録するフラグ

    public ViewerSeed_S()       //デフォルトのコンストラクタ
    {
        setuPLookAndFeel();  //ルック&フィールを設定する
        setupMenuBar();         //メニューを構築する
        setupToolBar();         //ツールバーを構築する
    }

    private void setuPLookAndFeel()
    {
        System.out.println(
            "System Look & Feel: "+ UIManager.getSystemLookAndFeelClassName());
                //システムの既定のルック&フィールをコンソールに出力
        //ルック&フィールを設定する。必要な行だけを生かして、あとはコメントにしておく
        try    {
//          UIManager.setLookAndFeel("com.sun.java.swing.plaf.motif.MotifLookAndFeel");
//          UIManager.setLookAndFeel("com.sun.java.swing.plaf.metal.MetalLookAndFeel");
                                                                //これは使えない
//          UIManager.setLookAndFeel("javax.swing.plaf.metal.MetalLookAndFeel");
                                                                //既定のルック&フィール
//          UIManager.setLookAndFeel("com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
//          UIManager.setLookAndFeel("com.sun.java.swing.plaf.mac.MacLookAndFeel");
                            //これを使うにはライブラリに mac.jar が必要
//          UIManager.setLookAndFeel("com.sun.java.swing.plaf.macos.MacOSLookAndFeel");
                            //これを使うにはライブラリに macos.jar が必要
        }           catch(Exception e)      {
                    System.out.println("...Use default Look and Feel");
                    System.out.println("...Missing..." + e.getMessage());
        }
    }

    private JMenuItem saveItem;     //「保存」の項目は状況によって選択できなくするため、
                                    //別のメソッドから利用できるようにメンバ変数にしておく

    private void setupMenuBar() //メニューを構築する
    {
        JMenuItem newItem, openItem, saveAsItem, closeItem, printItem, quitItem;
                                                                //「ファイル」メニューの項目
```

```
JMenuItem undoItem, cutItem, copyItem, pasteItem;       //「編集」メニューの項目
JMenuItem bgWhite, bgGray, bgYellow;
                                        //「編集」メニューの「背景色」のサブメニュー項目
JMenuItem aboutMRJ;                     //「ヘルプ」メニューに追加する項目

JMenuBar mb = new JMenuBar();           //メニューバーを用意する

/*「ファイル」メニューの作成*/
JMenu fileJMenu = new JMenu("ファイル", true);    //「ファイル」メニューを作成する
//      fileJMenu.setMnemonic(KeyEvent.VK_F);
                                        //ファイルメニューにニーモニックを指定する場合

newItem = new JMenuItem("New", KeyEvent.VK_N);       //「新規」項目を作成
newItem.setAccelerator(
        KeyStroke.getKeyStroke(KeyEvent.VK_N, ActionEvent. META_MASK ));
                                        //アクセラレーターキーは N
fileJMenu.add(newItem);                 //「ファイル」メニューに追加する
NewItemListener newListener = new NewItemListener();
        //「新規」を選択したときにイベントを受け付けるクラスを新たに生成
newItem.addActionListener(newListener);
        //「新規」を選んだときにイベントが発生するように設定

openItem = new JMenuItem("開く(O)...", KeyEvent.VK_O);      //「開く」項目を作成
openItem.setAccelerator(
        KeyStroke.getKeyStroke(KeyEvent.VK_O, ActionEvent. META_MASK ));
                                        //アクセラレーターキーは O
fileJMenu.add(openItem);                //「ファイル」メニューに追加する
OpenItemListener openListener = new OpenItemListener();
        //「開く」を選択したときにイベントを受け付けるクラスを新たに生成
openItem.addActionListener(openListener);
        //「開く」を選んだときにイベントが発生するように設定

closeItem = new JMenuItem("閉じる", KeyEvent.VK_W); //「閉じる」項目を作成
closeItem.setAccelerator(
        KeyStroke.getKeyStroke(KeyEvent.VK_W, ActionEvent. ALT_MASK ));
                                        //アクセラレーターキーは W
fileJMenu.add(closeItem);               //「ファイル」メニューに追加する
CloseItemListener closeListener = new CloseItemListener();
        //「閉じる」を選択したときにイベントを受け付けるクラスを新たに生成
closeItem.addActionListener(closeListener);
        //「閉じる」を選んだときにイベントが発生するように設定

fileJMenu.addSeparator();               //区切り線を追加する

saveItem = new JMenuItem("保存", KeyEvent.VK_S);      //「保存」項目を作成
saveItem.setAccelerator(
        KeyStroke.getKeyStroke(KeyEvent.VK_S, ActionEvent. META_MASK ));
                                        //アクセラレーターキーは S
saveItem.setEnabled(false);             //「保存」を選択できないようにイネーブルでなくす
fileJMenu.add(saveItem);                //「ファイル」メニューに追加する
saveAsItem = new JMenuItem("名前を付けて保存...");
                                //「名前を付けて保存」項目を作成、アクセラレーターキーはなし
fileJMenu.add(saveAsItem);              //「ファイル」メニューに追加する
SaveItemListener saveListener = new SaveItemListener();
```

```
            //「保存」「名前を付けて保存」を選択したときに
            //イベントを受け付けるクラスを新たに生成
saveItem.addActionListener(saveListener);
            //「保存」を選んだときにイベントが発生するように設定
saveAsItem.addActionListener(saveListener);
            //「名前を付けて保存」を選んだときにイベントが発生するように設定

fileJMenu.addSeparator();           //区切り線を追加する

printItem = new JMenuItem("印刷", KeyEvent.VK_P);      //「印刷」項目を作成
printItem.setAccelerator(
            KeyStroke.getKeyStroke(KeyEvent.VK_P, ActionEvent. META_MASK ));
                                    //アクセラレーターキーは P
fileJMenu.add(printItem);           //「ファイル」メニューに追加する
PrintItemListener printListener = new PrintItemListener();
            //「印刷」を選択したときにイベントを受け付けるクラスを新たに生成
printItem.addActionListener(printListener);
            //「印刷」を選んだときにイベントが発生するように設定

fileJMenu.addSeparator();           //区切り線を追加する

quitItem = new JMenuItem("終了", KeyEvent.VK_Q);       //「終了」項目を作成
quitItem.setAccelerator(
            KeyStroke.getKeyStroke(KeyEvent.VK_Q, ActionEvent. META_MASK ));
                                    //アクセラレーターキーは Q
fileJMenu.add(quitItem);            //「ファイル」メニューに追加する
QuitItemListener quitListener = new QuitItemListener();
            //「終了」を選択したときにイベントを受け付けるクラスを新たに生成
quitItem.addActionListener(quitListener);
            //「終了」を選んだときにイベントが発生するように設定

mb.add(fileJMenu);              //「ファイル」メニューをメニューバーに追加する

/*「編集」メニューの作成*/
JMenu editJMenu = new JMenu("編集", true); //「編集」メニューを作成する
EditItemListener editListener = new EditItemListener();
            //「編集」メニューを選択したときの処理を行うクラスを新たに生成する

undoItem = new JMenuItem("やり直し", KeyEvent.VK_Z);       //「やり直し」項目を作成
undoItem.setAccelerator(
            KeyStroke.getKeyStroke(KeyEvent.VK_Z, ActionEvent. META_MASK ));
                //アクセラレーターキーは Z
undoItem.setEnabled(false);
                //「やり直し」を選択できないようにイネーブルでなくす
editJMenu.add(undoItem);            //「編集」メニューに追加する
undoItem.addActionListener(editListener);
                //「やり直し」を選択したときにイベントが発生するようにしておく

editJMenu.addSeparator();           //区切り線を追加する

cutItem = new JMenuItem("カット", KeyEvent.VK_X);     //「カット」項目を作成
cutItem.setAccelerator(
            KeyStroke.getKeyStroke(KeyEvent.VK_X, ActionEvent. META_MASK ));
                                    //アクセラレーターキーは X
```

```
        editJMenu.add(cutItem);             //「編集」メニューに追加する
        cutItem.addActionListener(editListener);
                //「カット」を選択したときにイベントが発生するようにしておく

        copyItem = new JMenuItem("コピー", KeyEvent.VK_C);  //「コピー」項目を作成
        copyItem.setAccelerator(
                KeyStroke.getKeyStroke(KeyEvent.VK_C, ActionEvent. META_MASK ));
                                        //アクセラレーターキーは C
        editJMenu.add(copyItem);            //「編集」メニューに追加する
        copyItem.addActionListener(editListener);
                //「コピー」を選択したときにイベントが発生するようにしておく

        pasteItem = new JMenuItem("ペースト", KeyEvent.VK_V);       //「ペースト」項目を作成
        pasteItem.setAccelerator(
                KeyStroke.getKeyStroke(KeyEvent.VK_V, ActionEvent. META_MASK ));
                                        //アクセラレーターキーは V
        editJMenu.add(pasteItem);           //「編集」メニューに追加する
        pasteItem.addActionListener(editListener);
                //「ペースト」を選択したときにイベントが発生するようにしておく

        editJMenu.addSeparator();           //区切り線を追加する

        /*「編集」メニュー「背景色」でのサブメニューの作成*/
        JMenu bgColorJMenu = new JMenu("背景色");
                                            //サブメニューの元になる「背景色」項目を作成
        bgWhite = new JMenuItem("白");          //「白」項目を作成
        bgColorJMenu.add(bgWhite);              //「背景色」のサブメニューに追加する
        bgGray = new JMenuItem("グレイ");       //「グレイ」項目を作成
        bgColorJMenu.add(bgGray);               //「背景色」のサブメニューに追加する
        bgYellow = new JMenuItem("黄色");       //「黄色」項目を作成
        bgColorJMenu.add(bgYellow);             //「背景色」のサブメニューに追加する
        editJMenu.add(bgColorJMenu);
                                    //「背景色」のサブメニューを「編集」メニューに追加する

        BackgroundColorItemListener bgListener = new BackgroundColorItemListener();
            //「背景色」のサブメニューを選択したときの処理を行うクラスを新たに生成する
        bgWhite.addActionListener(bgListener);
            //「白」を選択したときにイベントが発生するようにしておく
        bgGray.addActionListener(bgListener);
            //「グレイ」を選択したときにイベントが発生するようにしておく
        bgYellow.addActionListener(bgListener);
            //「黄色」を選択したときにイベントが発生するようにしておく

        mb.add(editJMenu);    //「編集」メニューをメニューバーに追加する

        setJMenuBar(mb);      //メニューバーを Frame に設定する

        /*「ヘルプ」メニューへの追加*/
        JMenu helpJMenu = new JMenu("ヘルプ");    //「ヘルプ」メニューを作成する
        helpJMenu.add(aboutMRJ = new JMenuItem("MJR とは？"));
                                            //「MJR とは？」という項目を追加する
        mb.add(helpJMenu);                  //「ヘルプ」メニューをメニューバーに追加する
//      mb.setHelpMenu(helpJMenu);      //ここで作成したメニューをヘルプメニューとする
            /*      これにより、システムが用意する「ヘルプ」メニューと一体化する
```

```
                ただし、Swing1.1.1beta のドキュメントでは、このメソッドは組み込まれ
                ていないとなっている。
                メソッドがあればエラーでストップするようなので、とりあえずコメント
                化しておく                                */

        aboutMRJ.addActionListener(
                //ヘルプメニューに追加した項目を選択したときの処理を直接記述している
            new ActionListener()        {
                public void actionPerformed(ActionEvent ev)  {
                    System.out.println("Help");
                                //処理は単にコンソールに文字を出力するだけ
                }     }     );
    }

    private void setupToolBar()  //ツールバーの設定
    {
        JToolBar tb = new JToolBar();     //ツールバーを新しく作成する

        JButton newButton = new JButton("新規", new ImageIcon("icons/new.gif"));
                                        //ボタンを生成する。ボタン名は「新規」で、
                                        //指定した GIF ファイルのアイコンをつける
        newButton.setToolTipText("新しくウインドウを開く");   //チップヒントの文句
        tb.add(newButton);    //ボタンをツールバーに登録する
        NewItemListener newListener = new NewItemListener();
                //「新規」をクリックしたときにイベントを受け付けるクラスを新たに生成
        newButton.addActionListener(newListener);
                //「新規」をクリックしたときにイベントが発生するように設定

        JButton openButton = new JButton("開く", new ImageIcon("icons/open.gif"));
                                        //ボタンを生成する。ボタン名は「開く」で、
                                        //指定した GIF ファイルのアイコンをつける
        openButton.setToolTipText("ファイルを新しいウインドウで開く");
                                        //チップヒントの文句
        tb.add(openButton);   //ボタンをツールバーに登録する
        OpenItemListener openListener = new OpenItemListener();
                //「開く」をクリックしたときにイベントを受け付けるクラスを新たに生成
        openButton.addActionListener(openListener);
                //「開く」をクリックしたときにイベントが発生するように設定

        this.getContentPane().add(tb, BorderLayout.NORTH);
                //ツールバーをウインドウに登録する。
                //BorderLayout の機能を利用し、ウインドウの上端に配置する
    }

/*「ファイル」メニューの「新規」を選択、あるいは「新規」ボタンをクリックしたときの処理*/
    class NewItemListener implements ActionListener  {
                public void actionPerformed(ActionEvent ev)     {
            newDocumentByViewer();          //新規ドキュメントを用意する
        }
    }

/*「ファイル」メニューの「開く」を選択、あるいは「開く」ボタンをクリックしたときの処理*/
    class OpenItemListener implements ActionListener {
        public void actionPerformed(ActionEvent ev)  {
```

```
                openDocByViewerWithDialog(ViewerSeed_S.this);
                        //開くファイルをダイアログボックスで選択して実際に開く
        }
}

/*      実際に新規ドキュメントを用意する。
        ウインドウ外からも利用できるように static にしてある*/
static void newDocumentByViewer()       {
        new TextViewer_S(null);       //TextViewer_S のウインドウを新たに作成する
}

/*      ファイル選択ダイアログボックスを表示してファイルを選び、ビューアで開く。
        ウインドウ外からも利用できるように static にしてある*/
static void openDocByViewerWithDialog(Component parentWindow)          {
        File rootFolder = null;
        try     {
                rootFolder = MRJFileUtils.findFolder(new MRJOSType("docs"));
                                                                //「書類」フォルダへの参照を得る
        }               catch(Exception e)      {System.out.println(e.getMessage());            }
        JFileChooser fc = new JFileChooser(rootFolder);
                //初期フォルダを「書類」フォルダにしたファイル選択ダイアログボックスを作成
        fc.setFileFilter(new MacTextFileFilter());
                //テキストファイルだけを表示するフィルタを適用する
        int returnValue = fc.showOpenDialog(parentWindow);      //ダイアログボックスを開く
        if (returnValue == JFileChooser.APPROVE_OPTION)      //ファイルを選択したのなら
                openDocumentByViewer(fc.getSelectedFile());       //そのファイルを実際に開く
}

/*      指定したファイルを開くが拡張子に応じて利用するビューアを選択し、
        実際にそのビューアで開く。
        ウインドウ外からも利用できるように static にしてある*/
static ViewerSeed_S openDocumentByViewer(File targetFile)   {
        ViewerSeed_S instancedViewer = null;
        String fileNameLower = targetFile.getName().toLowerCase();
                //開くファイルのファイル名の小文字文字列を得る
        if(fileNameLower.endsWith(".htm") || fileNameLower.endsWith(".html"))
                instancedViewer = new HTMLViewer_S(targetFile);
                        //ファイルの末尾が.htm あるいは.html なら、HTML ビューアを用意する
        else
                instancedViewer = new TextViewer_S(targetFile);
                        //そうでなければ、テキストビューアを用意する
        return instancedViewer;       //生成したオブジェクトへの参照を戻す
}

/*「ファイル」メニューの「閉じる」を選択したときの処理*/
class CloseItemListener implements ActionListener {
        public void actionPerformed(ActionEvent ev)   {
                /*本来は修正されていれば保存する処理が必要*/
                dispose();          //Frame を閉じる
        }
}

/*「ファイル」メニューの「保存」「名前を付けて保存」を選択したときの処理*/
class SaveItemListener implements ActionListener  {
```

```
        public void actionPerformed(ActionEvent ev)  {
            String itemLabel = ((JMenuItem)ev.getSource()).getLabel();
                                                    //選択したメニューの項目名を取り出す
            if((openFile == null) || (itemLabel.compareTo("名前を付けて保存...") == 0))
    {
                                                    //選択したのが「名前を付けて保存...」なら
                File rootFolder = null;
                try    {
                    rootFolder = MRJFileUtils.findFolder(new MRJOSType("docs"));
                                                        //「書類」フォルダへの参照を得る
                }           catch(Exception e)     {System.out.println(e.getMessage());
    }
                JFileChooser fc = new JFileChooser(rootFolder);
                    //「書類」フォルダを初期フォルダにしたファイル選択
                    //      ダイアログボックスを生成
                int returnValue = fc.showSaveDialog(ViewerSeed_S.this);
                    //保存するファイルを指定するダイアログボックスを開く
                if (returnValue == JFileChooser.APPROVE_OPTION)     {           //保存する
なら
                    openFile = fc.getSelectedFile();    //保存するファイルを記録する
                    setTitle(openFile.getName());
                                //ファイル名をウインドウのタイトルにする
                    setDirty();  //変更したことを示すフラグを強制的に設定する
                }
                else
                    return;
            }
            if (isDirty)  {      //表示している中身に修正があれば
                doSave();          //それをファイルに保存する
                resetDirty();      //修正したことを示すフラグをクリアする
            }
        }
    }

    abstract void doSave();       //実際の保存処理は、継承したクラスで記述する

    /*「ファイル」メニュー印刷」を選択したときの処理*/
    class PrintItemListener implements ActionListener  {
        public void actionPerformed(ActionEvent ev)  {
            printDocumentByViewer();   //実際に印刷を行う
        }
    }

    /*     実際に印刷を行う。ウインドウ外からも利用できるように static にしてある*/
    void printDocumentByViewer()     {
        PrintJob pj = this.getToolkit().getPrintJob(this,"A",null);
                                //印刷設定のダイアログボックスが表示される
        Graphics pg = pj.getGraphics();    //印刷時のグラフィックス環境を取得
        if(pg != null)      {
            doPrint(pg);            //印刷メソッドを呼び出す
            pg.dispose();           //グラフィックス領域の破棄により、実際に印刷が始まる
        }
        pj.end();
    }
```

```
abstract void doPrint(Graphics pg);       //具体的な印刷作業は、継承したクラスで記述する

/*「ファイル」メニューの「終了」を選択したときの処理*/
class QuitItemListener implements ActionListener   {
        public void actionPerformed(ActionEvent ev)  {
                System.exit(0);   //アプリケーションを終了
        }
}

/*「編集」メニューを選択したときの処理*/
class EditItemListener implements ActionListener   {
        public void actionPerformed(ActionEvent ev)  {
                String itemLabel = ((JMenuItem)ev.getSource()).getLabel();
                                                        //選択したメニューの項目名を取り出す
                if(itemLabel.compareTo("やり直し") == 0)      //選択したのが「やり直し」なら
                        doUndo();  //実際の処理を行うメソッドを呼び出す
                else if(itemLabel.compareTo("カット") == 0)  //選択したのが「カット」なら
                        doCut();     //実際の処理を行うメソッドを呼び出す
                else if(itemLabel.compareTo("コピー") == 0)  //選択したのが「コピー」なら
                        doCopy();  //実際の処理を行うメソッドを呼び出す
                else if(itemLabel.compareTo("ペースト") == 0)//選択したのが「ペースト」なら
                         doPaste(); //実際の処理を行うメソッドを呼び出す
        }
}

abstract void doUndo();       //実際のやりなおし処理は、継承したクラスで記述する
abstract void doCut();        //実際のカットの処理は、継承したクラスで記述する
abstract void doCopy();       //実際のコピー処理は、継承したクラスで記述する
abstract void doPaste();      //実際のペースト処理は、継承したクラスで記述する

/*「編集」メニューの「背景色」から項目をを選択したときの処理*/
class BackgroundColorItemListener implements ActionListener        {
        public void actionPerformed(ActionEvent ev)  {
                String itemLabel = ((JMenuItem)ev.getSource()).getLabel();
                                                        //選択したメニューの項目名を取り出す
                if(itemLabel.compareTo("白") == 0)              //選択したのが「白」なら
                        setPaneBackground(Color.white);        //背景を白にする
                else if(itemLabel.compareTo("グレイ") == 0)  //選択したのが「グレイ」なら
                        setPaneBackground(Color.gray);         //背景をグレイにする
                else if(itemLabel.compareTo("黄色") == 0)     //選択したのが「黄色」なら
                        setPaneBackground(Color.yellow);       //背景を黄色にする
        }
}

abstract void setPaneBackground(Color c);
        //内部の色を設定する処理は、継承したクラスで記述する

/*      修正したことを記録し、「保存」メニューを使えるようにする。*/
public void setDirty()   {
        isDirty = true;
        saveItem.setEnabled(true);  //「保存」を選択できるようにする
}
```

```
    /*      修正した記録をクリアし、「保存」メニューは使えなくする*/
    public void resetDirty() {
        isDirty = false;
        saveItem.setEnabled(false); //「保存」を選択できないようにする
    }
}
```

MacTextFileFilter.java

JFileChooser でファイル選択のダイアログボックスを表示するときに、ここで定義した MacTextFileFilter をフィルタ関数として指定することで、一覧にはフォルダとテキストファイルだけが表示されるようにしています。同様な機能を第 6 章で、AWT の FileDialog 向けのフィルタクラスとして紹介していますが、その段階では、MRJ 2.1EA3 が利用できるバージョンでした。そのバージョンで、非表示ファイルかどうかをチェックする機能が、MRJ 2.1 正式版以降では使えなくなっており、別の方法に変わっています。ここでは、正式版での手法を利用します。ここで、ファイルの Finder 情報を取得する方法は分かったのですが、フォルダの情報を取得する方法が判明しませんでした。ファイルについては非表示ファイルはダイアログボックスには表示されないのですが、フォルダについては非表示のものも表示されてしまいます。

```
import java.io.*;
import javax.swing.filechooser.*;
import com.apple.mrj.MRJOSType;
import com.apple.mrj.MRJFileUtils;
import com.apple.MacOS.FSSpec;
import com.apple.MacOS.FInfo;          //MRJ2.1 正式版で初登場のクラス！？
//import com.apple.mrj.macos.generated.FinderConstants;     //MRJ2.1 正式版ではなくなった…

/*Swing の FileChooser で、テキストファイルとフォルダだけを表示するためのフィルタクラス*/
class MacTextFileFilter extends FileFilter    {
    public boolean accept(File f) {       //引数 f のファイルを表示するなら true を戻すメソッド
        boolean     returnValue = false;
        MRJOSType fType = null;

        if(f.isDirectory())  //フォルダは無条件にリストに含める
            /*      そのため、非表示フォルダも表示される。フォルダが非表示かどうかを
                求める機能が見つからない…        */
            returnValue = true;
        else  {       //以降、ファイルの場合
            try     {
                fType = MRJFileUtils.getFileType(f);      //ファイルタイプを取得する
                    /*デスクトップ管理の非表示ファイルなど、化け文字のあるファイル
                    では例外が発生してしまうものの、処理はストップしない。その意味
                    では、それらのファイルは自動的にリストに含めない。このエラーは、
                    化ける文字に関して Swing で得られたファイル文字列と、MRJ での
```

```
                        ファイル文字列が一致していないために起こると思われる。*/
                } catch(Exception e)    {
                        System.out.println(e.getMessage());
                        return false;       }
                            //例外が発生するファイルはともかくリストに含めない
                if(fType.equals(new MRJOSType("TEXT")))   //ファイルタイプが TEXT なら
                    returnValue = isNotVisible(f);
                            //非表示フラグがセットされていないならでリストに含める
                else  //ファイルタイプが TEXT でないなら、リストに含めない
                    returnValue = false;
            }
            return returnValue;
        }
        public String getDescription()      {
            return("Mac OS のテキストファイル");
        }

        /*  非表示フラグがセットされているかを調べる。されていなければ true を戻す。
            このメソッドはファイルにしか利用できない。
            また、化け文字が含まれるようなファイル（システムが作る隠しファイル）において、
            FInfo のコンストラクタでエラーが出る模様。例外ではなく、処理がストップしてしまう。
            たぶん、ファイル名の文字列がうまく処理されていないのだろうと思われる。
                99/3/21 MRJ2.1, Swing 1.1.1beta1*/

        private boolean isNotVisible(File targetFile)    {
            try{
                FSSpec targetSpec = new FSSpec(targetFile);
                                                    //FSSpec オブジェクトを構築する
                FInfo finderInfos = new FInfo(targetSpec);     //ファイルの Finder 情報を取得する
                if ((finderInfos.getFlags() & 0x4000) != 0)
                    //非表示のフラグをチェック。定数が定義されているクラスが見当たらない…
                    return false;       //フラグがセットされている
                else
                    return true; //非表示フラグがセットされていない
            }   catch(Exception e)
            {           return false;       }

    /*  //Swing の FileSystemView で非表示項目のチェックができるはずだが、機能していない。
        すべてのファイルで false を戻す
            FileSystemView fsv = FileSystemView.getFileSystemView();
            return(!fsv.isHiddenFile(targetFile));
        */
        }
    }
```

TextViewer_S.java

JTextArea を利用したテキストビューアのクラスです。メニューやツールバーの構築、アクションイベントの受付などは、ViewerSeed_S クラスで定義してしまっている

ので、このクラスでやっていることの半分以上は、JScrollPane と JTextArea を配置し、テキストファイルから読み出すところです。また、ファイルへの書き出しも作成しておきましたが、ファイルに書き出した後は、ファイルタイプの設定も行っています。

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import javax.swing.*;
import com.apple.mrj.*;

/*   ViewerSeed_S クラスを継承して、テキストビューアを定義する*/
public class TextViewer_S extends ViewerSeed_S
{
    private JTextArea viewText; //ウインドウ内に配置する TextArea

    public TextViewer_S(File targetFile)
    {
        openFile = targetFile;  //引数のファイルをメンバ変数に保存

        /*テキストを表示するための JTextArea と JScrollPane の用意*/
        viewText = new JTextArea();        //JTextArea を新たに用意する
        viewText.setLineWrap(true);
                                    //行の折り返しを行うようにする（動作は遅くなるが…）
        viewText.setWrapStyleWord(true);        //ワード単位の折り返しを行う
        viewText.addKeyListener(new textAreaKeyListener());     //キーイベントに反応させる
        JScrollPane scrollingPane = new JScrollPane(viewText);          //スクロール領域を用意
        scrollingPane.setPreferredSize(initilaWindowSize);
                                                    //スクロール領域の要求サイズを指定する
        scrollingPane.setHorizontalScrollBarPolicy(
            JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);
                                                    //水平スクロールバーを表示しない

        Container windowContents = getContentPane();
                                        //ウインドウの中身を管理するオブジェクトを取得
        windowContents.add(scrollingPane, BorderLayout.CENTER);
                                        //TextArea をウインドウに追加する

        /*ファイルから読み込んだ内容を TextArea に設定する*/
        if (openFile != null)     {       //ファイルが指定されているかをチェック
            setTitle(openFile.getName());       //ウインドウのタイトルはファイル名にする
            try    {
                FileReader LNR = new FileReader(targetFile);
                    //引数に指定したファイルから行読み込みする Reader を生成
                viewText.read(LNR, null);
                    //ファイルから読み取ったテキストを JTextArea に設定
                LNR.close();       //ファイルを閉じる
            }
            catch(Exception e)     {       //例外があったときの処理
                System.out.println(e.getMessage());}//エラーメッセージをコンソールに表示
        }
        pack();      //包含するオブジェクトのサイズにウインドウサイズを合わせる
```

```
        show();     //ウインドウを表示する
}

/*      キーを押したときに発生するイベントを処理するクラスを定義       */
class textAreaKeyListener implements KeyListener {
        public void keyTyped(KeyEvent e){       //キーが押されたとき
                setDirty();  //表示内容を修正したことにする
        }
        public void keyPressed(KeyEvent e)      {       }
        public void keyReleased(KeyEvent e)     {       }
}

void doUndo()   {       /*実装せず…*/           }
void doCut()    {       viewText.cut();         }       //カットの処理
void doCopy()   {       viewText.copy();        }       //コピーの処理
void doPaste()  {       viewText.paste();       }       //ペーストの処理

void setPaneBackground(Color c) //背景色の設定
{
        viewText.setBackground(c); //背景色を設定し
        viewText.repaint();                     //再描画
}

void doSave()   {       //保存の処理を記述する
        if (openFile != null)     {       //ファイルが指定されているかを一応チェック
                try     {
                        FileWriter fWriter = new FileWriter(openFile);
                                //ファイルへ書き込みをする FileWriter を生成する
                        viewText.write(fWriter);        //JTextArea の内容をファイルに書き込む
                        fWriter.close();    //FileWriter を閉じる
                        MRJFileUtils.setFileTypeAndCreator(
                                openFile, new MRJOSType("TEXT"), new MRJOSType("MJR1"));
                                        //作成したファイルのタイプとクリエイターを設定する
                }
                catch(Exception e)      {       //例外があったときの処理
                        System.out.println(e.getMessage());}//エラーメッセージをコンソールに表示
        }
}

void doPrint(Graphics g)      {       //印刷の処理を記述する
        String targetLine;              //1 行分を保存する変数
        int currentBase = 0;            //ベースラインを計測する変数
        int LinePitch = 16;             //行ピッチを設定
        StringTokenizer eachLines = new StringTokenizer(viewText.getText(), "¥n", false);
                //JTextArea の内容を改行で区切ったトークンに分解する
        while(eachLines.hasMoreTokens())      {       //トークンを順番に取得
                targetLine = eachLines.nextToken();     //現在のトークンを得る
                currentBase += LinePitch;    //ベースラインを行ピッチ分進める
                g.drawString(targetLine, 0, currentBase);       //行を描画する
                        /*      単に描画しているだけで、用紙幅で折り返すようなことはしていない。
                                ページ送りもしていない      */
        }
        /*      以下のメソッド呼び出しだけでも印刷できる。その場合、ウインドウ内の JTextArea
                に表示されているような改行位置で折り返されるなどするが、ページ制御はしてい
```

```
            ないので、テキストすべてが印刷されるわけではない。
        viewText.print(g);
        */
    }
}
```

HTMLViewer_S.java

JEditorPane を利用した HTML ビューアのクラスです。これもテキストのビューアと同様に、ViewerSeed_S クラスを継承して作っています。また、表示だけをすることを意図しているので、コンストラクタで必要なコンポーネントを用意するだけにしてあります。

```
import java.awt.*;
import java.io.*;
import javax.swing.*;
import com.apple.mrj.*;

/*  ViewerSeed_S クラスを継承して、HTML ビューアを定義する*/
public class HTMLViewer_S extends ViewerSeed_S
{
    JEditorPane viewHTML;      //HTML エディタのオブジェクト

    public HTMLViewer_S(File targetFile)
    {
        openFile = targetFile;  //引数のファイルをメンバ変数に保存

        /*テキストを表示するための JEditorPane と JScrollPane の用意*/
        try   {
            viewHTML = new JEditorPane("file://" + targetFile.getPath());
                        //指定したファイルを表示する JEditorPane を新たに用意する
        }           catch(IOException e)  {      System.out.println(e.getMessage());    }
        JScrollPane scrollingPane = new JScrollPane(viewHTML);      //スクロール領域を用意
        scrollingPane.setPreferredSize(initilaWindowSize);
                                                    //スクロール領域の要求サイズを指定する
        scrollingPane.setHorizontalScrollBarPolicy(
            JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);
                                                    //水平スクロールバーを表示しない

        Container windowContents = getContentPane();
                                        //ウインドウの中身を管理するオブジェクトを取得
        windowContents.add(scrollingPane);    //JEditorPane をウインドウに追加する

        setTitle(targetFile.getName());      //ウインドウのタイトルはファイル名にする

        pack();      //包含するオブジェクトのサイズにウインドウサイズを合わせる
        show();      //ウインドウを表示する
    }
```

```
    /*      各種の処理は省略しました…        */
    void doUndo()    {       }
    void doCut()     {       }
    void doCopy()    {       }
    void doPaste()   {       }
    void setPaneBackground(Color c) {       }
    void doSave()    {       }
    void doPrint(Graphics g)     {       }
}
```

LOCUS Report

第8章　Mac OS で使う Swing

● 機能豊富な GUI コンポーネントを利用できる Swing は、Java で手軽に高機能なソフトウエアを作成することにもつながり大変に注目されています。

● Swing は Java2 に標準で組み込まれました。しかしながら、JDK1.1.X 上でもライブラリを追加することで利用できます。

● もちろん、Mac OS 上でも利用できます。MRJ 2.1 を利用した上で Swing を利用したプログラム作成について説明をしましょう。

● Swing は、ある意味では基本コンポーネントである AWT の拡張です。この章では、AWT でのプログラミングとの違い、そして Mac OS で動かすプログラムでの注意点を中心に解説を行います。

8-1　Mac OS で使う Swing

Swing についての基本的なことと、Mac OS で Swing を使って開発するために必要な作業についてまとめておきます。

Swing の特徴

ウインドウやメニューのような GUI に欠かせないものに加え、テキストボックスやボタンなどを始めとするコントロール類もよく利用されます。古くはそうしたコントロールを独自に設計して作り、それを利用するような時代もありました。しかしながら、現在はシステム機能としてコントロールを豊富にサポートし、プログラムをする側は、手軽にたくさんのコントロールを使うということを要求するようになっています。Windows の Visual Basic のように、最初からたくさんのコントロールが利用でき、さらに追加することもできるため、コントロール類だけを販売するようなビジネスまで成り立つようになってきています。Mac OS では、Ver.8.0 で、コントロールの種類を大幅に増やし、かなりのバリエーションのコントロールが、指定するだけで利用できるようになりました。

こうしたコントロールを使うメリットは、もちろん、手軽にプログラム作成できるという点にあります。豊富なコントロールがあれば、手軽に高い機能のソフトウエアを作ることにもつながります。コントロールは GUI 部品に限りませんが、GUI 部品として使われるものが多くなっています。

Java の基本ライブラリでは、AWT の中で、コントロールを当初からサポートしてきていました。しかしながら、利用できるコントロールは基本的なものに限られていました。そこで、JFC（Java Foundation Class）として、豊富なコントロールをサポートするようになり、JDK 1.1.X の時代に開発が進められてきました。Java2 では JFC は標準コンポーネントとなっています。Swing は JFC のもっとも重要な要素で、豊富な GUI 部品を提供するライブラリです。JFC で提供される大部分は Swing であるとも言って良いくらいで、注目が集まったためにコードネームとしての Swing が正式名称となってしまったと言ってよいでしょう。

♦機能が豊富な GUI コンポーネント

Swing の注目できるところは機能が豊富なところで、つまり多種多様なコントロールがサポートされているということでしょう。AWT までしか使えなかった時代では、

Java の機能不足が何かにつけて取りざたされていましたが、基本的なコントロールしか使えない状態ではそれも否定できない状態でした。しかしながら、Swing はそれを打破したと言えるでしょう。現在、一般的なアプリケーションに使われているようなさまざまなコントロールは、ほとんどが Swing によってプログラムで利用できるようになります。いずれにしても、Java で作るプログラムのユーザーインタフェースを多彩にする極めて重要なコンポーネントが Swing であると言えるでしょう。

用意されているコンポーネントは、非常にたくさんあります。目に付く点としては、まずツールバーが簡単に構築できるという点があります。また、メニューやボタンは非常に多くの表示バリエーションを持っています。テキスト関連のコントロールでは、単なるテキストエディタだけでなく、スタイル付きのエディタやその発展系として HTML のエディタまであります。もちろん、単にコンポーネントをウインドウに貼り付ければエディタを作ることができるというわけではないものの、表示するだけであれば、コンポーネントの配置だけで可能でもあり、その意味でも豊富で高い機能の一端が表現できるのではないでしょうか。表形式のテーブルコントロールも利用できます。また、テキストだけでなく多くのコントロールでグラフィックスも利用できるなど、見栄えのするユーザーインタフェースを作成するというレベルまでの機能を持っています。

Swing はコントロール群というだけでなく、ユニークな機能も見られます。まず、ルック&フィールとしてコンポーネントの見かけやあるいは諸設定をクラスとして定義することが挙げられます。これにより、ソフトウエア全体の見かけを、使用するクラスの取り替えで簡単に変えられるということになります。標準的なルック&フィールが用意されていますが、Windows 風、Macintosh 風などが用意されており、OS の使用感に近付けることもできます。また、独自のカスタマイズもできると言えるでしょう。つまり、Mac OS で言えばアピアランスあるいはテーマのような機能が利用できるわけです。

♦Java で構築されているライブラリ

Swing 自体は、すべての部分が Java で作られているため、ライブラリ自体はプラットフォームに依存するものではありません。つまり、Java が稼動していれば、Swing が必ず利用できると言えるわけです。AWT を基礎にして、Java だけで構築しているため、Swing 自体のクロスプラットフォームの問題は比較的出にくい状況にあるとも言えるでしょう。

実用面から見た Swing

機能が高くなれば、重くなることは避けられません。実用面でどうなのかを検討してみましょう。

まず、Swing のライブラリ自体のファイルは 2MB 程度で、そこそこのサイズです。アプリケーションといっしょに配付するとなると、ちょっと大きなサイズであるかなとも思えるかもしれませんが、MRJ の場合、Java 標準クラスだけでも 7MB を超えるサイズであることを考えれば、むしろこれほどの機能の割には小さいとも言えるでしょう。また、バージョンアップはあるものの、Java2 では標準に組み込まれる機能なので、配付についての心配も、特に将来的には少なくなると思われます。

Swing の動作については、やはりただでさえ遅いことが指摘される Java の上で、その Java を使って作られたライブラリだけに、やはり重たい物になっています。もちろん、ネイティブな機能を使ってシャキシャキ動く GUI ライブラリという方向性も短期的には実用的かも知れませんが、こうしたライブラリ自体を Java で作るというのが、Java の 1 つの進む道筋を表しているとも言えます。マシンの性能が加速度的に向上している現在では、重いと思っていたソフトもいつの間にか軽くなっていることもあります。少し先を見た場合には、重たいという問題の比重も低くなるのではないかと思います。

メモリについてもかなり食います。Mac OS で、1M 程度のアプリケーションメモリを確保していても、Swing を使うとそのアプリケーションメモリが何と 10M くらいまで一気に広がります。メモリ自体のコストも安くなったとは言え、概してブラウザ並に 1 つのアプリケーションがメモリを消費するというのは、現状のアプリケーションの水準からしてかなり多くのメモリを使うと言わざるを得ないでしょう。配付して使うようなアプリケーションの場合には、それなりに配慮は必要なレベルだと思われます。

また、Mac OS 環境ということから見た場合、完全に OS 機能と統合されていない面はあります。具体的に次の節で説明をしますが、Swing はまだ発展途上にあると言えるような段階です。もちろん、Ver.1.1 まで正式版が出ており、それなりに完成度は高くなっていますが、概して、完成したと結論付けるにはまだ早い段階だと思います。もういくらかもまれる必要があると考えられます。

Mac OS 環境では、MRJ に Swing が組み込まれていないこともあり、Swing によるソフト開発は、状況にもよりますが、まだ時期尚早な面もなくはないでしょう。すぐ

に利用したい実用ソフトでは、スペックの低いマシンで限定された場合などは利用をあきらめるという場面も出てくると思われます。一方、将来を見越した場合には、Java の世界でも必須の機能だと言えます。マシン環境が良くなるなどの環境の変化を考えれば、Swing は将来的には大きく有望な機能であることは間違いないことと言えるでしょう。

Swing ライブラリの組み込み

Mac OS で Swing 対応のアプリケーションなどを開発する場合に必要な準備を説明しましょう。まず、ここでは、MRJ 2.1 および MRJ SDK 2.1 の正式リリース版をインストールしているものとします。

MRJ 2.1 は、対応する JDK のバージョンが 1.1.6 であり、Swing を含まない Java の実行環境です。そのため、Swing を別途入手する必要があります。Swing 自体は販売されているわけでなく、以下のアドレスより無償で入手できます。

http://java.sun.com/products/jfc/

MRJ 2.1 上で開発するときには、JDK 1.1 対応の JFC を入手します。1999 年 3 月末のこの原稿を執筆している段階では、Swing 1.03、Swing 1.1、Swing 1.1.1β1 の 3 種類の JFC がリリースされていました。ともかくいちばん新しい、Swing 1.1.1β1 を使ってサンプルプログラムを作成しました。

ダウンロードは上記の Web ページで操作をして行いますが、使用する OS を選択するようになっています。ここで、Mac OS で開発するのでれば、Mac OS 向けのものをダウンロードすると良いでしょう。すると、Mac OS のアプリケーション形式のインストーラがダウンロードできます。開発キットと言えば、ファイルを圧縮しただけのようなものがむしろ多いのですが、MRJ SDK などと同様、インストーラでインストールができます。ただし、基本的にはファイルを展開するだけのようです。ちなみに、インストーラは InstallAnywhere を使って作られたもののようです。

図 8-1　ダウンロードしたインストーラ

インストーラの利用方法は難しいことはありません。単にダブルクリックして、以下のようなウィザード形式の質問に答えて行けばそれでファイルがハードディスクに展開されます。以下、インストーラのポイントになる部分だけをかいつまんで説明をしておきましょう。

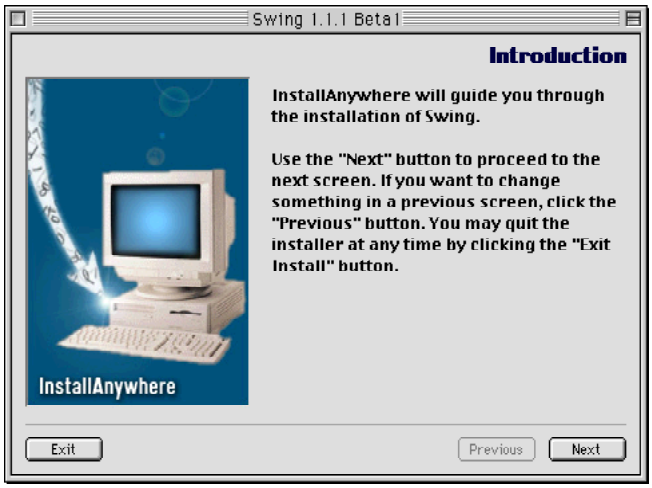

図 8-2　ウィザード形式のインストーラ

インストール作業の途中で、次のように、インストール先のフォルダを指定する画面になります。ここで、Choose を選択して、ファイルを選んでもかまいません。既定値では「アプリケーション」フォルダになっています。日本語のフォルダ名を選択しても機能します。

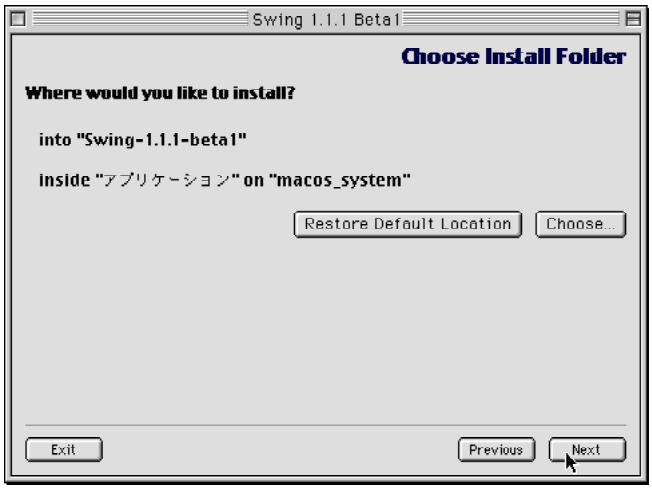

図 8-3　保存する先を指定する

インストーラの途中では次のように、エイリアスをどこに作成するかを選択する画面になります。この中に、何のエイリアスなのかが明確に書かれていないので分かりづらいのですが、これは、Swing のサンプルプログラムのアプリケーションへのエイリアスです。デスクトップにサンプルのエイリアスがあると、もちろん、サンプルを即座に試すことができるので便利と言えば便利ですが、サンプルをしょっちゅう起動するということもないと思われるので、不要なら Don't create aliases を選択しておくとよいでしょう。

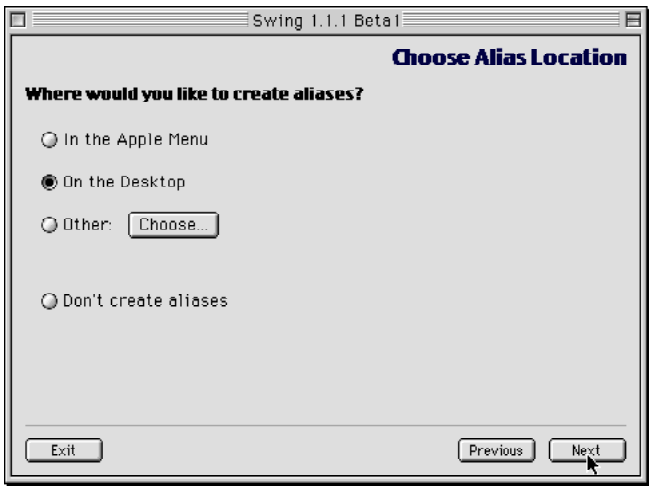

図 8-4　サンプルアプリケーションのエイリアスのインストール先を指定

さらに、次のように、セットアップする内容を選択する画面になります。開発を行うのであれば、通常はいちばん下の Full Developer Release を選択することになるでしょう。クラスライブラリの使い方を記したドキュメントは、これを選択しないとイン

ストールされません。

図 8-5　インストールする内容は、Full Developer Release を選択する

こうしてインストールが完了します。まず、サンプルアプリケーションへのエイリアスをデスクトップに作成するように指定したので、次の図のように、アプリケーションのデフォルトアイコンで、アプリケーションへのエイリアスが作られています。SwingSet というのが Swing のコンポーネントを一覧にして見せるアプリケーションで、よく引き合いに出されるものです。アイコンはデフォルトになっていますが、ダブルクリックすると起動して更新され、Java のキャラクターがデザインされたアイコンに変わります。

図 8-6　デフォルトのアプリケーションアイコンがデスクトップに配置されたサンプルのエイリアス

ここで、実際に Swing 対応のアプリケーションが機能するかをチェックすれば良いでしょう。SwingSet というエイリアスをダブルクリックすると、Swing のコンポーネントにどのようなものがあって、Mac OS では実際にどんなふうに見えるかがわかるので、これを実行してみれば良いでしょう。

また、インストール先のフォルダとしてここでは「アプリケーション」フォルダを選択しましたが、インストール先には「Swing-1.1.1-beta1」というフォルダが作られていて、その中に、次のようにファイルが展開されています。これらのファイルを開発あるいは実行時に利用します。

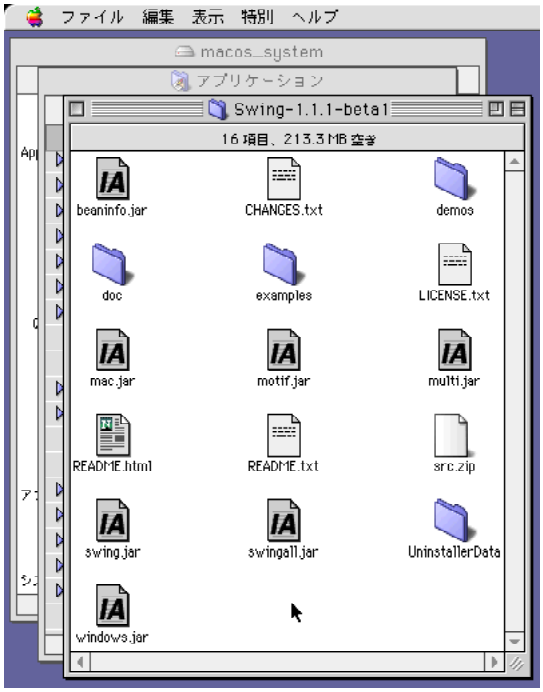

図 8-7　インストールされたフォルダの中身

ここで、フォルダの中にいくつかファイルが展開されていますが、拡張子が.jar のいくつかあるファイルが Swing の本体です。通常は Swingall.jar というファイルを利用します。使用するファイルについてはこの後で紹介します。なお、いずれのファイルもアーカイブされたままの状態で使用します。

表 8-1　インストールによって得られるライブラリファイル

ファイル名	内容
swing.jar	Swing のクラスライブラリ本体。ルック&フィールは Metal だけが含まれる
windows.jar	Windows のルック&フィールを納めたファイル

motif.jar	Motif のルック&フィールを納めたファイル
beaninfo.jar	Beans に関連するファイル
swingall.jar	swing.jar, windows.jar, motif.jar, beaninfo.jar の内容を 1 つにまとめたファイル
mac.jar	Mac のルック&フィールを納めたファイル
multi.jar	ルック&フィールを複数利用するための機能を提供するファイル

あと、README.html が Netscape Navigator の文書で含まれています。テキストファイルが Mac OS の改行形式になっていないなどの理由から、添付文書はいずれもブラウザで見る事になります。README.html を開けば、基本的な注意書きだけでなく、フォルダに含まれる各種文書へのリンクもあります。また、ライブラリの使い方のドキュメントへのリンクもあります。

プログラムで Swing を利用する

Swing を利用したプログラムを CodeWarrior で作る場合の一般的な手順を示しておきましょう。個別の内容として次の節で説明することもありますが、ここでは一般的な内容を説明しておきます。

まず、Java のアプリケーションあるいはアプレットを作成するためのプロジェクトを用意します。Java のカテゴリにあるいずれかのプロジェクトのひな形を選択して、保存するフォルダを指定するのは通常通りです。以下は、Java Application を指定した場合を説明します。

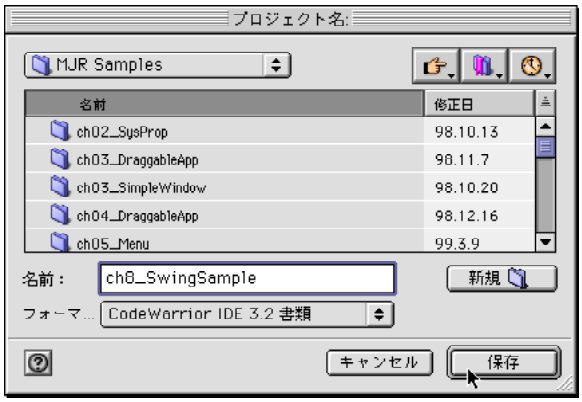

図 8-8　プロジェクトのファイル名とフォルダ名を指定する

こうして作られたプロジェクトのフォルダ（この例では、ch8_SwingSample）には、

プロジェクトのファイルと、アプリケーションのひな形となる TrivialApplication.java のファイルが作られているはずです。

まず、作られたプロジェクトのフォルダ内に Swing の本体である Swingall.jar ないしは Swing.jar ファイルをコピーしておきます。

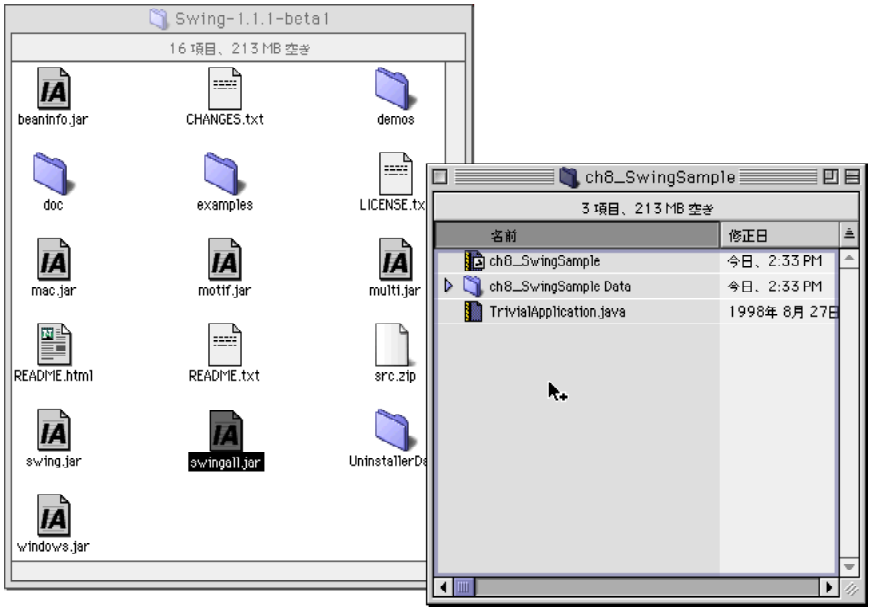

図 8-9　Swingall.jar をプロジェクトのフォルダにコピーする

また、MRJ の基本機能だけを使うのであれば、MRJ SDK の MRJToolkit フォルダにある MRJToolkitStub.zip もプロジェクトのフォルダにコピーしておくと便利でしょう。

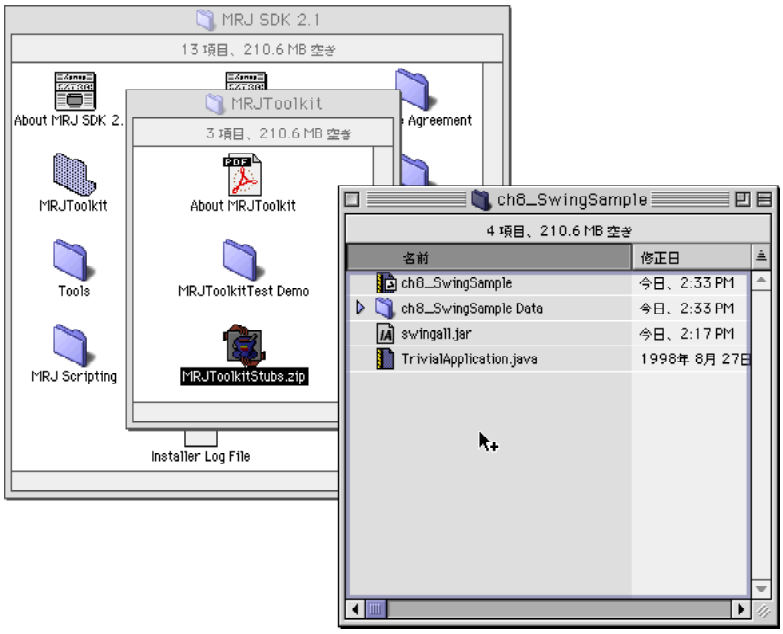

図 8-10　MRJToolkitStub.zip もコピーしておく

そして、Swingall.jar、MRJToolkitStub.zip を、プロジェクトのウインドウにドラッグ&ドロップして、プロジェクトに追加しておきます。

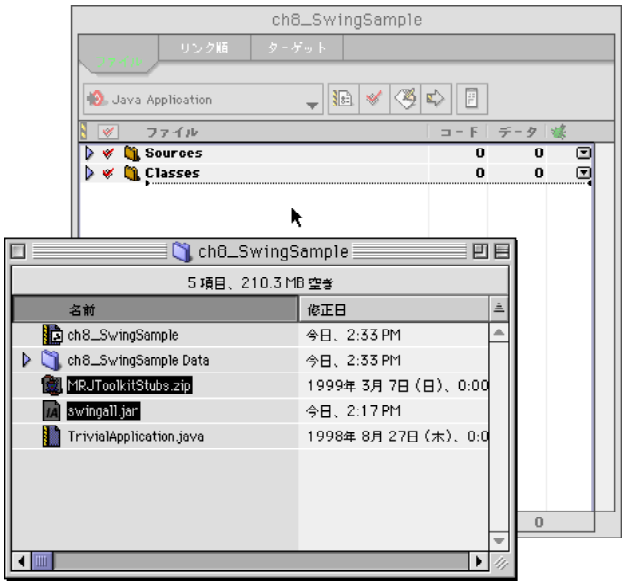

図 8-11　フォルダにコピーしたファイルをプロジェクトに登録する

ただし、この MRJToolkitStub.zip では、MRJ のすべての機能は利用できません。もし、MRJToolkit の機能を超えたプログラムを作成するには、システムフォルダの「機

能拡張」フォルダにある、MRJLibraries フォルダの MRJClasses フォルダにある MRJClasses.zip をプロジェクトに登録しておきます。Finder からドラッグ&ドロップしてもかまいませんが、「プロジェクト」メニューから「ファイルを追加」を選択して、機能拡張フォルダ内の MRJClasses.zip を選択して登録しても良いでしょう。

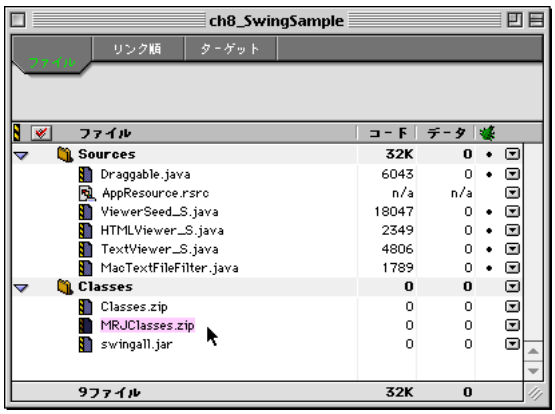

図 8-12　プログラムによっては、MRJClasses.zip をプロジェクトに登録する

なお、この章のサンプルプログラムは、MRJClasses.zip をプロジェクトに登録しておく必要があります。

8-2　Swing の機能を使う

Mac OS 上で Swing の機能を使った場合に留意すべき点などをまとめておきましょう。Swing は非常に広大なライブラリで、すべてのクラスについてチェックをしたわけではなく、サンプルプログラムの範囲内でのチェックとなりますが、ウインドウやメニューなどの基本的なことはチェックしてあります。

ルック&フィールの使い方

ボタンのデザインやあるいはメニューのデザインなど、GUI 部品の見かけをルック&フィールとして選択することができるのが Swing の大きな特徴です。Swing に最初からいくつかのルック&フィールが含まれていますが、ルック&フィール自体を Java で開発することができるので、場合によっては独自のものを作るということも可能です。実際、Mac OS 向けに別のルック&フィールがリリースされています。

まず、基本セットで利用できるルック&フィールを見てみましょう。swing.jar ライブラリに含まれている Metal という名前のルック&フィールは、Swing のデフォルトのルック&フィールと言える存在で、「特に指定しなければ」（ただし、この表現はやや間違いがあります）このルック&フィールを利用すると言えば良いでしょうか。文字通り金属的なイメージはしなくもないですが、ともかく、OS に関係なく利用することを意図した基本ルック&フィールのようです。

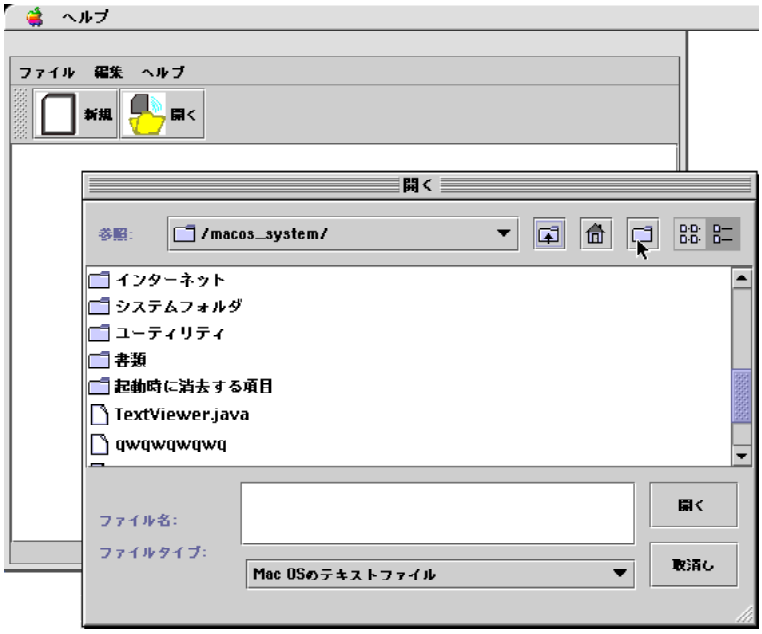

図 8-13　Metal のルック&フィールを選択したときの様子

ルック&フィールを設定する機能は、Swing のライブラリの中にある UIManager というクラスを利用します。このクラスにあるスタティックメソッド setLookAndFeel を使って、ルック&フィールの構築に使用するクラス名を文字列で指定します。

```
import javax.swing.*;
    :
UIManager.setLookAndFeel("javax.swing.plaf.metal.MetalLookAndFeel");
```

引数にある、javax.swing.plaf.metal.MetalLookAndFeel というのは、Metal ルック&フィールの処理を行うクラス名の完全な記述です。この MetalLookAndFeel というクラスは、swing.jar あるいは swingall.jar に含まれています。なお、setLookAndFeel メソッドは、例外の処理を組み込まなければなりません。

ライブラリとして、swingall.jar、あるいは swing.jar と windows.jar を使っていれば、Windows 風のルック&フィールにすることもできます。たとえば、ファイル一覧ではフォルダのアイコンが黄色だったり、ツールバーやダイアログボックスの色合いなどが、Windows 風になります。

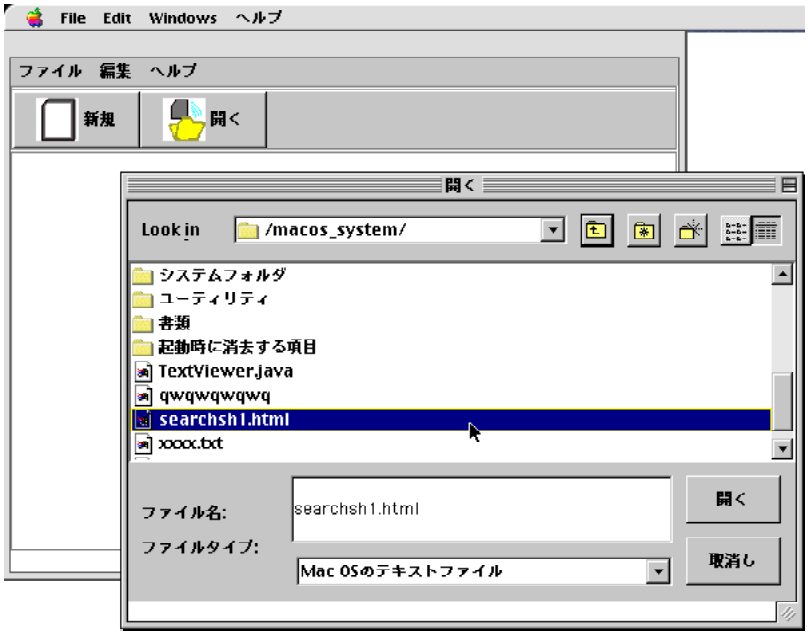

図 8-14　Windows **のルック&フィールで表示した**

Windows のルック&フィールに設定するには、やはり、setLoolAndFeel メソッドを使います。たとえば、次のような 1 つの命令があれば、それで Swing 全体のルック&フィールが Windows 風に設定されます。

```
import javax.swing.*;
    :
UIManager.setLookAndFeel(" com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
```

Windows のルック&フィールを提供するクラスは、com.sun.java.swing.plaf.windows.WindowsLookAndFeel となっており、階層が Metal と大きく異なっています。これは、Swing の Ver.1.0 時代に、すべてのクラスが、com.sun.java.swing 以下の階層で定義されていたなごりでしょう。Ver.1.1 から Javax の階層に変わったのですが、ソース中にクラス名の全階層を記述しているためにバイトコードにその文字列が含まれることになります。そこで、過去との互換性のために、ルック&フィールのクラスはそのまま com.sun…になっているのだと思われます。

同様に、Motif 風のルック&フィールを提供するクラスもあります。これは、swingall.jar を使っているか、swing.jar と motif.jar を使っている場合に利用できます。setLookAndFeel の引数に、com.sun.java.swing.plaf.motif.MotifLookAndFeel という文字列を指定すると、Motif 風のルック&フィールになります。

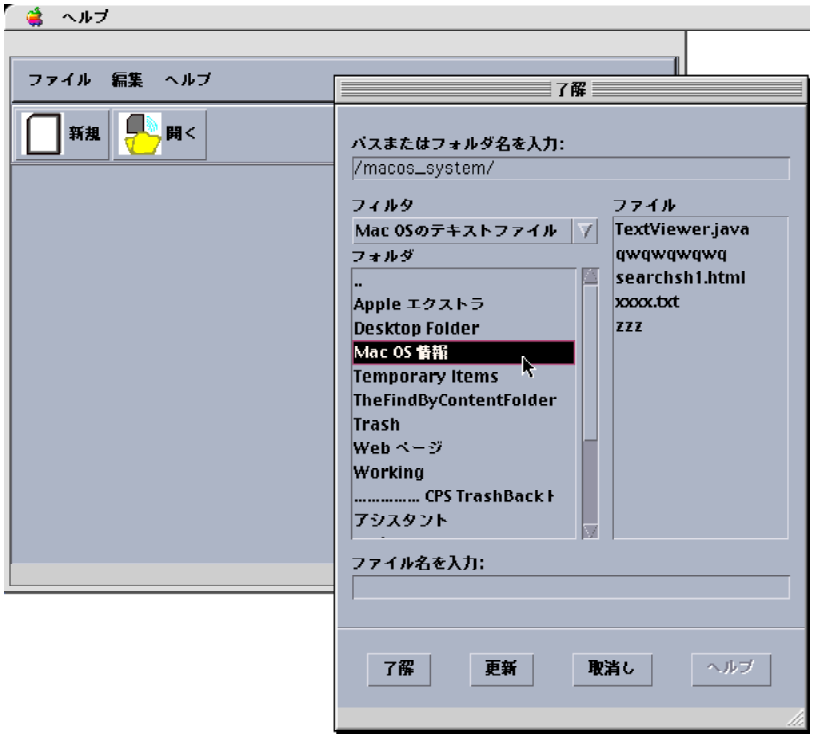

図 8-15　Motif **風のルック&フィールに設定した**

♦Mac OS 風のルック&フィールを使う

Swing には、Mac OS 風のルック&フィールも用意されています。ただし、このルック&フィールを実現するクラスは、swingall.jar には含まれていません。そこで、swingall.jar あるいは swing.jar のいずれを使う場合にでも、mac.jar はプロジェクトに含めなければなりません。mac.jar ファイルは、Swing によってインストールされる一連のファイルの中に含まれているはずです。

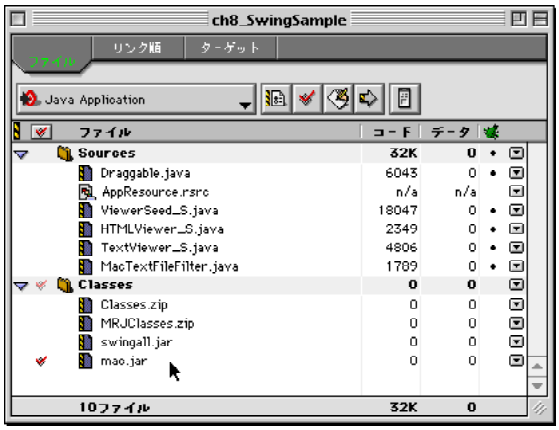

図 8-16　Mac OS **のルック&フィールを使うには、**mac.jar **をプロジェクトに含める**

プロジェクトに、mac.jar が含まれた状態で、次のように、setLookAndFeel メソッドを使うと、ルック&フィールは Mac OS 風になります。

```
import javax.swing.*;
    :
UIManager.setLookAndFeel("com.sun.java.swing.plaf.mac.MacLookAndFeel");
```

図 8-17　Mac OS のルック&フィールで表示した

Mac OS のルック&フィールを使った時には、そのときのアピアランスの設定も反映されます。この図は、ラージシステムフォントに Osaka Bold を指定したときのものなので、メニューが太字になっています。ただし、ウインドウに登録したメニューは、メニューバーではなく、ウインドウのタイトルバーのすぐ下に表示されており、その意味では Windows 的です。メニューについての問題は、別のところで検討しましょう。

また、図ではファイル選択ダイアログボックスが表示されています。これは、システムのファイル選択ダイアログボックスというよりも、Swing でカスタマイズ可能なファイル選択ダイアログボックスの機能があり、それを利用して Mac OS 風にしたという感じのものになっています。どうせ真似るなら、Navigation Service のダイアログボックスを真似てほしいとも感じるところです。

♦システム標準のルック&フィール

使用するルック&フィールを指定しない場合には、自動的に Metal が使用されます。

つまり、Metal がルック&フィールのデフォルトになっています。

一方、現在のシステムのルック&フィールを得る機能が Swing には用意されています。UIManager クラスにある getSystemLookAndFeelClassName メソッド（スタティック）を利用すると、現在のシステムに適したルック&フィールのクラス名を文字列で得られます。たとえば、次のような命令があれば、コンソールにシステムのルック&フィールのクラス名をコンソールに出力します。

```
import javax.swing.*;
    :
System.out.println("System Look & Feel: "+ UIManager.getSystemLookAndFeelClassName());
```

Mac OS 上で実行すると、getSystemLookAndFeelClassName メソッドの戻り値は、com.sun.java.swing.plaf.mac.MacLookAndFeel つまり Mac OS のルック&フィールになります。

注意したいのは、getSystemLookAndFeelClassName メソッドの戻り値は設定されている値ではなく、あくまで Swing によって推薦されるクラスだというところです。これで得られたクラスでルック&フィールを構築するには、setLookAndFeel クラスで改めて設定をしてやらなければなりません。逆にそうしたプロセスを組み込めば、実際に稼動している OS に近いルック&フィールを自動的に選ぶということも可能になります。

ところが、Mac OS のルック&フィールのクラスは、swing.jar はもちろん swingall.jar にも含まれていません。getSystemLookAndFeelClassName メソッドが、Mac OS のルック&フィールのクラスを戻したからと言って、必ずしもそのクラスが利用可能な状態になっているとは限らないのです。つまり、mac.jar をプロジェクトに必ず含めておかないといけないということになります。

Swing でのウインドウ

Swing のウィンドウに関する機能は、AWT のウインドウである Frame クラスと基本的にはほとんど同じように使えます。むしろ、共通に使えるメソッドが多いので、Swing だからと言って異なる点はほとんどありません。たとえば、表示や非表示はもちろん、AWT と同様のレイアウト機能を利用することができ、また、メニューなどの登録もできるので、基本的には AWT と同じように利用すればそれで済みます。

javax.swing.JFrame、つまり JFrame クラスが定義されており、たとえば、このクラ

スを継承することによって、Swing に対応したウインドウを作成できます。ウインドウは、その時に指定されたルック&フィールに従った見かけで表示されます。

♦Frame と JFrame の違い

Frame と JFrame の違いの 1 つは、コンポーネントの追加処理にあります。Frame 自体が Container のサブクラスであったため、ボタンなどのコンポーネントを Frame に対して add メソッドで追加できました。つまり、ウインドウである Frame に直接追加できたわけです。

一方、JFrame は直接の追加はできません。JFrame にある getContentPane メソッドを使って、ウインドウの中のコンポーネント追加可能な場所を取得し、そこに対して add メソッドでコンポーネント類を追加するようにします。プログラムは、AWT では直接 add していたところを、getContentPane メソッドを通じて得られたオブジェクトに対して add するように、一段階プロセスが増えるだけです。getContentPane の戻り値の型は AWT の Container です。

♦ウインドウのクローズは自動的にできる

Frame で作ったウインドウは、そのままではクローズボックスをクリックしてもウインドウは閉じられませんでした。そこで、イベントリスナを組み込んでクローズボタンをクリックしたことをクラスに伝えられるようにし、そのイベントを受けてウインドウをクローズしたり、あるいはアプリケーションをクローズするようなプログラムを組む必要がありました。

しかし、Swing の JFrame では、そうしたイベント処理は組み込む必要はありません。クローズボックスのクリックにより、自動的にウインドウは閉じられます。もちろん、閉じるときに何らかの処理を行うのであれば、イベントを取り扱うようにする必要はありますが、単に閉じるだけでいいのであれば、何の処理も組み込む必要はなくなったわけです。

Swing でのメニュー

メニューについては、やはり AWT に近い形式のクラスが Swing でも定義されていますが、いろいろな意味で違ってきます。また、Mac OS で動かすアプリケーションとなると、さらに考慮すべきことが出てきます。メニューについては Swing 独自の事情を認識しておかなければなりません。

まず、Swing のメニューを利用するための基本的なクラスを表にまとめておきまし

た。基本的には、AWT のクラス名に J が頭に付くと考えれば良いでしょう。

表 8-2　メニューで利用するクラス

構成要素	AWT のクラス	Swing のクラス
メニューバー	MenuBar	JMenuBar
メニュー	Menu	JMenu
メニュー項目	MenuItem	JMenuItem
ショートカット	MenuShortcut	(KeyStroke)
選択処理	ActionListener	ActionListener

メニューバーやメニュー、メニュー項目を生成して、それらを必要な要素に登録すれば、ウインドウのメニューは作成されます。メニューバーをウインドウに登録するには、JFrame で定義された setJMenuBar メソッドを利用すれば良いでしょう。

ちなみに、AWT の機能で作られたメニュー処理プログラムにおいて、Menu を JMenu に一括置換するだけで、基本的なところは全部置き換わり、Swing 対応のメニュー処理プログラムになります。ただし、違いのある部分のプログラム修正は必要になりますが、まずは一括置換をするというのが第一歩になるでしょう。

♦メニューを Mac OS のメニューバーに表示する

Swing のメニューは、基本的には必ずウインドウのタイトルバーの下に、ウインドウごとに表示されます。つまり、Windows などの GUI の形式なので、Mac OS のようなメニューバーにメニューを出すような形式にはなっていません。これを、Mac OS 的にする方法はなくはありません。

1 つは、Swing のメニューを使わないことです。JFrame を使っていても、AWT のメニュー処理でメニューを構築すると、MRJ 環境下では、メニューバーを使ったメニューが表示されます。その場合、メニューバーのウインドウへの登録は、JFrame に対して setMenuBar を使います。JFrame は Frame を継承しており、AWT のメニューに関連する機能は、Frame で定義されたメソッドを使うことになります。ただし、Swing のメニュー機能は非常に豊富で、メニューに文字やショートカットだけでなく、アイコンなどのグラフィックスやさらにはコントロールのようなものまで配置できます。そうした豊富な機能を望むのであれば、Swing を使い、ウインドウ内に表示されるメニューを使わざるを得ないでしょう。

もう 1 つの方法は、Sun Microsystems ではないところで開発された Mac OS L&F という独自のルック&フィールを使うことです。このルック&フィールを使えば、Swing のメニュー関連の機能を使っても、Mac OS のメニューバーにメニューが表示されま

す。ただし、Mac OS のメニュー機能に依存する面もあるので、やはり使えない処理も出てきて、それなりに制約があります。このルック&フィールについては、別にまとめておきましょう。

♦メニュー項目の生成とショートカットやアクセラレーターキー

メニューを組み立てる基本は、AWT と同様です。ただし、ショートカットキーの扱いは大きく違います。AWT のメニューの場合、1 つの方法として MenuItem のインスタンスを生成するときに、引数に MenuShortcut クラスのインスタンスを指定していました。それによってショートカットキーを割り当てるわけです。

一方、Swing では、用語の使い方が異なります。Windows ではメニューを選択するのに、たとえば Alt キーを押して、F キーを押して、O キーを押すと、「ファイル」メニューの「開く」を選択するような操作体系があります。それに利用するのを「キーボードニーモニック」と呼んでいます。一方、Ctrl+X や Command+X のように、キーボード処理一発で何かメニュー項目を呼び出す機能は「アクセラレーターキー」と呼んでいます。

話がややこしくなりますが、Windows では、Ctrl+X などのキー操作はショートカットと呼んでおり、Alt→F→O のような順序での F や O キーの事を、「アクセスキー」と呼んでいます。このあたりは用語が混乱しますが、以後は Swing 用語で説明することにします。

たとえば、メニューバーを作成し、メニューを作成し、さらにメニュー項目を作成する部分を抜粋すると次のようになります。

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

    :

JMenuItem newItem

JMenuBar mb = new JMenuBar();      //メニューバーを用意する

/*「ファイル」メニューの作成*/
JMenu fileJMenu = new JMenu("ファイル", true); //「ファイル」メニューを作成する

newItem = new JMenuItem("新規", KeyEvent.VK_N);    //「新規」項目を作成
newItem.setAccelerator(
          KeyStroke.getKeyStroke(KeyEvent.VK_N, ActionEvent. META_MASK ));
                                                  //ショートカットは N
fileJMenu.add(newItem);           //「ファイル」メニューに追加する
NewItemListener newListener = new NewItemListener();
```

```
                    //「新規」を選択したときにイベントを受け付けるクラスを新たに生成
newItem.addActionListener(newListener);
                    //「新規」を選んだときにイベントが発生するように設定
    :
    :

/*「ファイル」メニューの「新規」を選択、あるいは「新規」ボタンをクリックしたときの処理*/
class NewItemListener implements ActionListener      {
                public void actionPerformed(ActionEvent ev)  {
        newDocumentByViewer();          //新規ドキュメントを用意する
    }
}
```

まず、JMenuBar を生成してメニューバーを作成します。さらに JMenu を生成してメニューを作成しますが、1 つ目の引数はメニュー項目名、2 つ目の引数はティアオフメニューかどうかを示すのですが、現在はまだ機能は組み込まれていません。

そして、JMenuItem が実際のメニュー項目に相当します。生成するときのコンストラクタで 2 つの引数を取っていますが、1 つ目がメニュー項目名、2 つ目が int 型ならニーモニックキーを指定します。ちなみに、2 つ目が javax.swing.ImageIcon 型のアイコン画像なら、アイコン付きのメニュー項目を生成できます。

ニーモニックキーがどのキーかを指定するためには、java.awt.event.KeyEvent クラスに定義されたスタティック変数を使用します。このクラスには、VK_で始まるキーコードを記述した変数が定義されているので、それを利用して、キーを指定します。キーと変数の対応は、API のドキュメントを参照してください。たとえば、VK_N は、「N」のキーを示しています。なお、JMenu にニーモニックキーを指定する場合には、setMnemonic メソッドを使用し、引数に KeyEvent クラスのスタティック変数を指定します。

このニーモニックキーを使ったメニュー選択は、Metal のルック&フィールのときは、F10 キーを押します。そうすれば、この章のサンプルプログラムだと「ファイル」メニューが開きます。そこで、たとえば「N」キーを押すと、「新規」が選択されるという具合です。各ルック&フィールでのキー操作については、setLookAndFeel で指定するクラスの API ドキュメント中にリンクがあり、そこをクリックすると一覧表で参照できるようになっています。いきなりメニューが開くあたりは、意図的かどうか、Windows との違いを出しています。ただ、ニーモニックキーについては完全に日本語化されていない面があります。英語だと、New という項目のニーモニックキーが N だとしたら、New という単語の N の部分に下線が引かれます。しかしながら、「新規」というメニュー項目に N をニーモニックキーと指定しても、N がニーモニックであるという情報はメニュー項目には表示されません。Windows では、「新規(N)」のような

記述をしてニーモニックに相当するアクセスキーを指定していますが、日本語だとそうしたメニュー名を文字列として与えないといけなくなります。

そして、アクセラレーターキーは、生成した JMenuItem のインスタンスに対して、setAccelerator メソッドを使用して設定します。このメソッドの引数には、javax.swing.KeyStroke を指定しますが、このクラスに KeyStroke を生成するスタティックなメソッド getKeyStroke を使用します。このメソッドはオーバーロードされていて、引数に対して複数のバリエーションがあります。しかしながら、メニューのアクセラレーターキーは一般には、文字キーと修飾キーの組み合わせになるでしょう。1 つ目に文字キーを、2 つ目に修飾キーを指定します。文字キーはニーモニックキーと同様、java.awt.event.KeyEvent のスタティック変数を指定します。そして、修飾キーは、java.awt.Event クラスのスタティック変数を利用します。Ctrl キーは CTRL_MASK、Shift キーは SHIFT_MASK になります。Command キーは META_MASK になります。option キーに対応する機能はないようです（ALT_MASK で ALT キーを指定してもだめでした）。複数の修飾キーを指定する場合には各変数を加えればよいでしょう。このあたり、まじめにマルチプラットフォームソフトを作る場合には、OS によってアクセラレーターキーを異なるものに設定することになるでしょう。

こうしてアクセラレーターキーの指定を行えば、そのキー操作により自動的にメニュー項目に指定したイベント処理が行われます。つまり、キー操作を受け付けるイベント操作を定義する必要はありません。

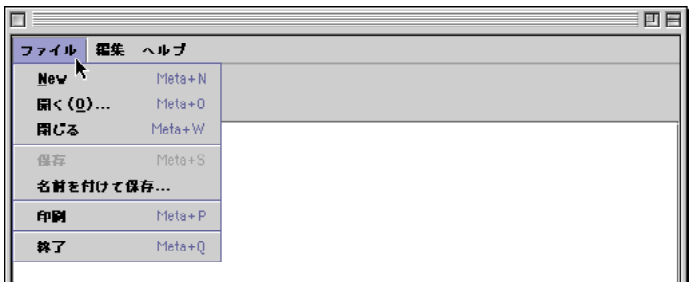

図 8-18　作成したメニューの一例

作成したメニューを見てみると、まず、Command キーなのに、メニュー上のアクセラレーターキー表示が、Meta となっていることです。これは、Mac OS の標準のルック&フィールを使っていても同様です。Command キー独特のマークが出てこないと、Mac OS の利用者は不満に思うところでしょう。

また、別の問題があります。テキスト編集可能な JTextArea をウインドウ内に配置

してある場合、たとえば、カットしようとして Command+X を押すと、「x」という文字が入力されてしまうのです。Command キーを押した状態でキーボードを押すと、どれもキー入力されてしまうので、アクセラレーターキーなら入力されないようにするような処理がどこかで欠けているのではないかと思います。これをプログラムで回避するのはかなり複雑そうですし、第三者が作った Mac OS 向けのルック&フィールではこの点は解決されています。この点は、Swing 自体で修正されるのを待った方が良いと思われます。

ヘルプメニューに関する処理が AWT のメニューに組み込まれています。Swing にもメソッドは用意されているのですが、まだ機能が組み込まれていないという風に API ドキュメントで解説されています。ヘルプメニューへの追加は、現在のバージョンでの Swing では行わないようにしておく必要があります。

♦MacOS L&F を使う

Luca Lutterotti 氏が、Swing に付属する Mac OS 向けのルック&フィールをもとに改良したものを配付しています。Swing オリジナルのものを Mac L&F と呼び、Luca Lutterotti 氏のものは区別するために、Mac OS L&F と同氏は呼んでいます。

Mac OS L&F の大きな特徴は、JMenuBar で作成したメニューは、ウインドウのタイトルバーの下ではなく、Mac OS のデスクトップとしてのメニューバーにきちんと表示されることです。また、Command キーのショートカットは、きちんと“クローバーマーク”でメニューに表示されます。単にルック&フィールを切り替えるだけで、より Mac OS に適合できるのです。こうしたルック&フィールを開発していることは賞賛できることですが、このようなプラグイン的な機構を持っている Swing の柔軟性も注目できるところでしょう。

この Mac OS L&F は、Luca Lutterotti 氏の Web ページである「Java On Mac」（http://www.ing.unitn.it/~luttero/javaonMac/）から入手できます。いくつかのパターンがありますが、ルック&フィールを実現するクラスを含む、macos.jar というファイルが利用できれば良いでしょう。

ただし、この Mac OS L&F を使うためには、パッチの当たった Swing のライブラリである swingall.jar を利用しなければなりません。実際、Swing 1.1.1β1 の swingall.jar を利用するようにしていると、正しく機能しません。

プロジェクトで Mac OS L&F を利用するには、たとえば、次のように作業します。まず、Java On Mac のページからダウンロードしたファイルを解凍します。ここでは「SwingSetMacOS」というフォルダに展開したとします。その中にある swingall.jar と

macos.jar の 2 つのファイルを、作成中のアプリケーションのフォルダ（ch08_SwingSample）にコピーします。コピー先にオリジナルの swingall.jar がある場合には、たとえばフォルダを作っておくなどして隠しておけばよいでしょう。

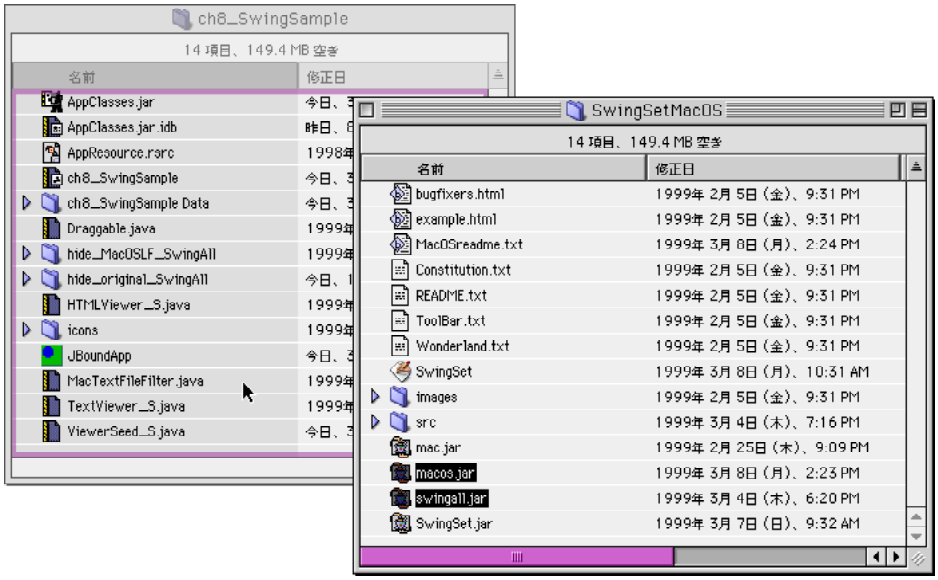

図 8-19　macos.jar、swingall.jar をコピーする

そして、それらコピーした 2 つのファイルをプロジェクトに登録します。念のため、いったん swingall.jar をプロジェクトから削除し、その上で、swingall.jar と macos.jar をプロジェクトのウインドウにドラッグ&ドロップして登録すればよいでしょう。

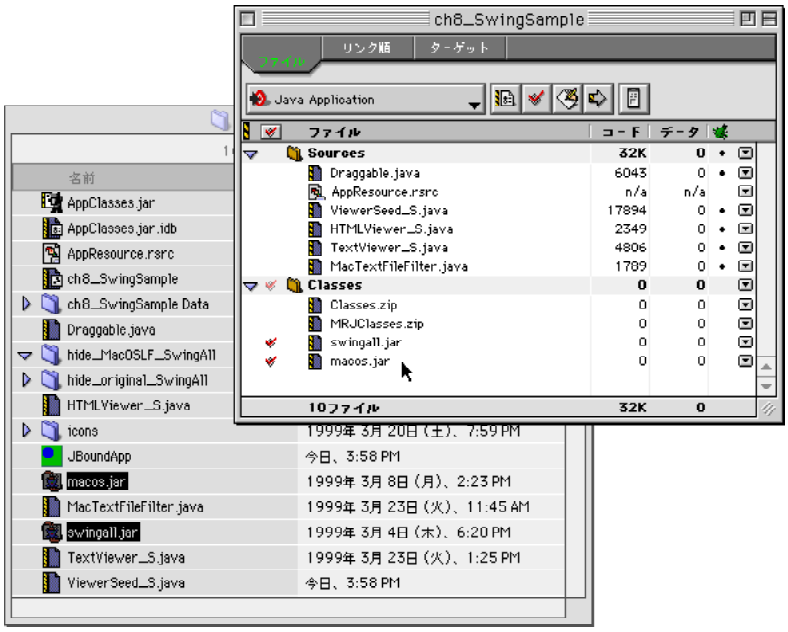

図 8-20　macos.jar、swingall.jar **をプロジェクトに登録する**

なお、swingall.jar を、Mac L&F のものとオリジナルの物で切り替えて使うには、それぞれフォルダに隠ししたり出したりすればいいのですが、その都度プロジェクトへ登録しなおす必要はないようです。CodeWarrior は、アクセスパスで認識するフォルダにある swingall.jar ファイルを取り込むので、エイリアスなどを使用しているわけではなく、あくまでファイル名で認識しているようです。従って、必ずしもプロジェクトへの登録しなおしは必要はないというわけです。

Mac OS L&F を使うには、以下のような命令を実行します。これによって、システム全体のルック&フィールが Mac OS L&F になります。Mac OS L&F を利用するためのクラスは以下のプログラム中の引数文字列を参照してください。実際にはさらに例外処理が必要になります。

```
UIManager.setLookAndFeel("com.sun.java.swing.plaf.macos.MacOSLookAndFeel");
```

こうして Mac OS L&F を使って表示させたアプリケーションを見てみると、メニューはきちんと Mac OS のメニューバーに表示されています。また、アクセラレーターキーの表示も、Mac OS 的にクローバーマークとアルファベットで表示されています。

図 8-21　Mac OS L&F **で表示した**

ただし、メニューの区切り線が表示されません。JMenu にある addSeparator メソッドが機能していないようです。また、システムのヘルプメニューに追加する機能もドキュメントでは説明されていますが、英語の Help という名称のメニューでないといけないようで、日本語システムのヘルプメニューへの追加もできないようです。

このように、まだ不完全だとは言え、Swing をより Mac OS 的にするルック&フィールとして、Mac OS L&F は注目しておく必要があるでしょう。

ツールバー

Swing にはツールバーのためのクラスが用意されている点が、AWT とは大きな違いです。ツールバーを管理する JToolBar クラスがあり、ツールバー内にはさまざまなコンポーネントを配置できます。一般には、JButton クラスで作るボタンを配置するのが手軽でしょう。JButton をインスタンス化するときに、引数を 2 つ指定し、1 つ目にボタンに表示する文字、2 つ目にアイコンを指定すれば、アイコンと文字の含まれたボタンが作成されます。アイコンは、IconImage クラスを利用しますが、これもコンストラクタの引数に GIF ファイルへのパスを指定すれば、その GIF ファイルをアイコンのデザインとしたアイコンを生成できます。また、JButton には、setToolTipText メソッドを利用して、ツールチップ、つまりマウスポインタをボタンの上に移動させたときに、ボックスが出てきてそのボタンの機能説明などを行うことができます。これも、メソッド 1 つだけで実現できます。

ボタンをクリックしたときに何か処理をする場合には、メニューとまったく同様に、アクションイベント処理を組み込みます。ActionListener をインプリメントしたクラスを呼び出すような仕組みを利用します。

最終的には、JToolBar を、JFrame の中に配置します。JFrame のコンテナを getContentPane メソッドで得て、それに add メソッドでツールバーを追加します。このあたりは、他のコンポーネントと同様に扱います。

以下はサンプルプログラムの一部です。JButton の引数にあるように、「icons/new.gif」のようなパスを指定しますが、これにより、アプリケーションのあるフォルダの icons フォルダにある new.gif という画像ファイルをアイコンに設定することができます。相対パスの場合には、アプリケーションの存在するフォルダが基点になるようです。JFrame のコンテナでは、BorderLayout の機能を利用しているため、NORTH の位置にツールバーを配置すると、ちょうどタイトルバーやメニューバーが表示されるすぐ下のよくある位置にツールバーが配置されることになります。

```
import javax.swing.*;
    :
JToolBar tb = new JToolBar();   //ツールバーを新しく作成する

JButton newButton = new JButton("新規", new ImageIcon("icons/new.gif"));
        //ボタンを生成する。ボタン名は「新規」で指定した GIF ファイルのアイコンをつける
newButton.setToolTipText("新しくウインドウを開く"); //チップヒントの文句
tb.add(newButton);          //ボタンをツールバーに登録する
NewItemListener newListener = new NewItemListener();
        //「新規」をクリックしたときにイベントを受け付けるクラスを新たに生成
newButton.addActionListener(newListener);
```

```
        //「新規」をクリックしたときにイベントが発生するように設定
    :
this.getContentPane().add(tb, BorderLayout.NORTH);
    //ツールバーをウインドウに登録する。BorderLayout の機能を利用し、
    //      ウインドウの上端に配置する
```

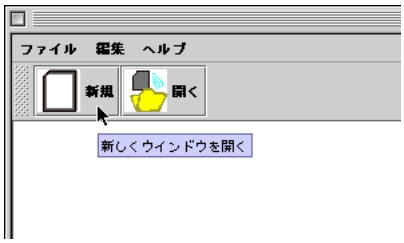

図 8-22　ウインドウに追加したツールバー。ツールチップを表示させた

なお、JToolBar は、特に何もプログラムを組まなくても、ドラッグして移動することができます。最初の状態のツールバーの左端は、ざらざらしたデザインになっていますが、ここをドラッグすることでフローティングツールバーになります。また、ツールバーの左端をドラッグして、ウインドウの上下左右の端に移動させると、その位置でウインドウにツールバーがドッキングします。なお、フローティングウインドウにした場合、常に手前に表示されるパレット形式にはならず、ウインドウの背後に隠れてしまうこともあるので、そのあたりの操作では注意が必要になります。

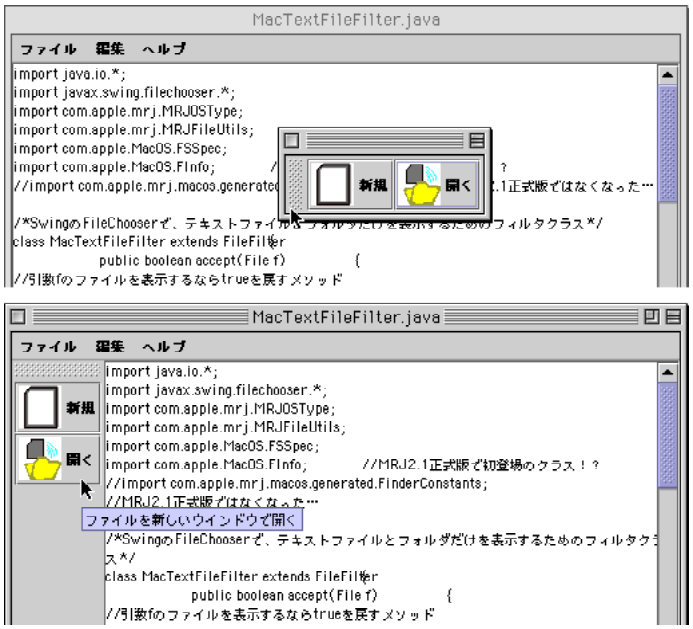

図 8-23　フローティングやあるいは左に配置したツールバー

Swing のコンポーネント

この章のサンプルプログラムでは、テキスト表示の JTextArea と HTML 表示の JEditorPane を使ったビューア的なアプリケーションを組んでみました。それらを利用した範囲内にはなりますが、Swing でのコンポーネントの様子をまとめておきましょう。

♦テキストを表示する JTextArea

テキストを開いて表示する場合は、AWT だと TextArea を使っていましたが、Swing ではその上位互換コンポーネントとでも言える JTextArea があります。こうしたコンポーネントは、AWT の場合は単にウインドウやアプレットに追加すればよかったのですが、Swing の JFrame では、まず、getContentPane メソッドを使って、コンポーネントを追加できるコンテナを得ます。さらにテキスト領域をスクロールさせたい場合には、スクロール領域の JScrollPane クラスのインスタンスを作成する必要があります。つまり、テキスト領域の JTexArea を生成し、それを含む JScrollPane を生成する必要があります。その JScrollPane を、getContentPane で得られた JFrame のコンテナに add メソッドで追加します。以上の手順が、スクロール可能なコンポーネントの基本的な追加手順です。つまり、スクロールとテキスト表示が TextArea ではまとまっていたのが、Swing では別々のコンポーネントで処理をするということになります。

JTextArea は、TextArea と違って、ウインドウ幅で折り返すような表示も可能になっています。setLineWrap(true)で折り返しするようになり、setWrapStyleWord メソッドで、ワード単位の折り返しをするかどうかの指定もできます。ただし、折り返しをすると処理速度は目に見えて落ちるくらいになります。

以上のように、テキストデータを表示する JTextArea があって、その一部分をスクロールすることで見せる JScrollPane があります。たとえば、BorderLayout がレイアウト機能として利用されている場合には、JScrollPane をその Center 領域に追加します。

そして、これらのいくつかのコンポーネントの大きさを調整しなければなりません。ウインドウのサイズを変更したときの大きさ調整は、BorderLayout を利用していれば自動的になされます。レイアウト機能を使わない場合には、独自にサイズ調整が必要になるでしょう。プログラム上でサイズ調整をしなければならないのは、ウインドウを作成した最初の段階です。いろいろやってみた結論として、まず、最終的に JFrame の pack メソッドを使って最終調整をすることになるのですが、そこで望む大きさにウインドウがなるように、この場合だと JScrollPane のサイズを調節します。単にサイ

ズを調節するのではなく、setPreferredSize メソッドを使って、望ましいサイズ（Prefered Size）の設定を行います。pack メソッドは望ましいサイズに設定するメソッドです。それで、ウインドウに含まれる JScrollPane を設定します。そして、JTextArea については、自動的にサイズが設定されるようです。おそらく、pack メソッドを実行した段階で、ウインドウに含まれるコンポーネントの全体的なサイズ調整が行われるのではないかと思われます。

JTextArea に文字列を設定するには、TextArea と同様、append メソッドなどが使えますが、この章のサンプルプログラムのようにテキストファイル、一般にはストリームからの入力が分かっているのであれば、read メソッドで、ストリームから読み込んだテキストをテキスト領域に設定します。そのとき、Reader クラスを指定するので、ファイルから読み取るのであれば、FileReader クラスを生成して、それを指定すると、FileReader クラスが管理するテキストファイルの内容を JTextArea に取り込みます。同様に、JTextArea の中身を書き出すには、Writer クラスを指定して write メソッド 1 つでストリームに書き出します。ファイルに書き出すには、FileWriter を生成します。これらのメソッドは、Mac OS 上で実行した場合、Mac OS 向けの動作をします。read は改行コードの種類に関係なく読み取りを行うようです。write は、Mac OS では CR コードだけを書き出します。

JTextArea で非常に便利なのは、クリップボード処理がほとんど 1 つのメソッドでできることです。文字通り、cut、copy、paste のメソッドがあり、クリップボードのデータの取得などをプログラミングする必要はありません。また、これらのショートカットとして、Ctrl+X、C、V は自動的に働くようになっています。ただし、Command+X、C、V は、メニュー処理のショートカットとして呼び出さないといけません。メニューのところで指摘したように、メニューのショートカットとしては機能するものの、副作用として、X、C、V の文字まで入力されてしまいます。このあたりは、Swing 側の対応を待つのが良さそうです。

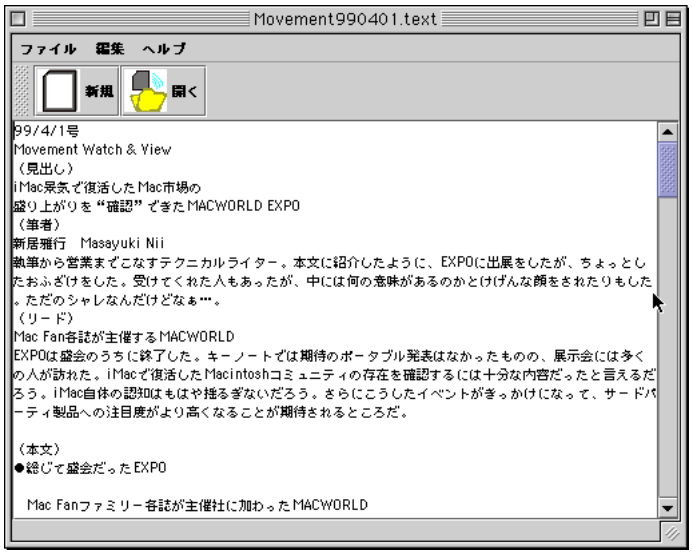

図 8-24　**テキストファイルの内容を表示した** JTextArea

♦HTML **エディタを作成できる** JEditorPane

Swing の JEditorPane クラスを使えば、HTML テキストの表示に加え、内容の編集もできます。ただし、文字の編集程度ならともかく、タグが絡むような編集作業は単に配置しただけではすべては行えないのですが、メニューやツールバーに機能を用意すれば、これで HTML エディタ並のワープロが簡単に作れるのではないかと期待してしまいます。JEditorPane を使うだけで、リンク先もテキストとして編集できます。リンク先をクリックしてジャンプするブラウザの機能にすることも可能で、リンクをクリックしたというイベント処理を組み込みます。この章のサンプルではそこまでの機能は含めていません。

JEditorPane の使い方は、JTextArea と基本的には同じです。JEditorPane のコンストラクタの引数に URL を指定すれば、その URL の中身を表示する JEditorPane が作成されるので、read による読み込みも場合によっては必要ありません。

以下の図は、GoLive CyberStudio3 で作った HTML ページのファイルを、JEditorPane で表示した例です。完全に HTML を表示しているのではなく、タグはタグとして表示する場合もあるようです。いずれにしても、HTML エディタを簡単に作れるというあたりに非常に可能性を感じるというところでしょうか。

図 8-25　CyberStudio **で作ったページを** JEditorPane **で表示した**

♦ファイル選択ダイアログボックス

Swing にはファイル選択のダイアログボックスも、javax.swing.FileChooser というクラスで提供されています。AWT とは別の機能となっていて、どちらかと言えば、Windows のファイル選択ダイアログボックスに近いデザインになっています。開くや保存の処理はもちろん、カスタマイズしたダイアログボックスも表示できます。前にも説明したように、Mac OS 向けのルック&フィールでは、このファイル選択ダイアログボックスは、従来システムの標準ダイアログのデザインを、さらに真似して作ったような使い勝手の悪いものになっています。Metal などのルック&フィールに比べても機能的に削られている面もあり、残念なところです。ダイアログボックスを Swing の機能で生成していることもあるのでしょうが、いずれにしても、Navigation Service の機能は使えません。一方、MJR 2.1 以降では、AWT の FileDialog クラスを使えば、Navigation Service 対応のダイアログボックスが表示されます。実用性を考慮すれば、現状では Swing のファイル選択ダイアログボックスは使わないで、AWT の機能を利用する方がベターな気がします。

また、Swing のファイル選択ダイアログボックスでは、フォルダの階層がどう見ても余分に付きます。フォルダ階層のドロップダウンリストを見れば、フォルダ階層にある最後に、フルパスの階層が付いています。ファイル一覧には、漢字など日本語は正しく表示されますが、スラッシュが%2f で表示されるなど、Finder で見えるファイル名とは必ず一致していません。つまり、Java のシステムとして都合のいいファイル名の文字列になっているというわけです。

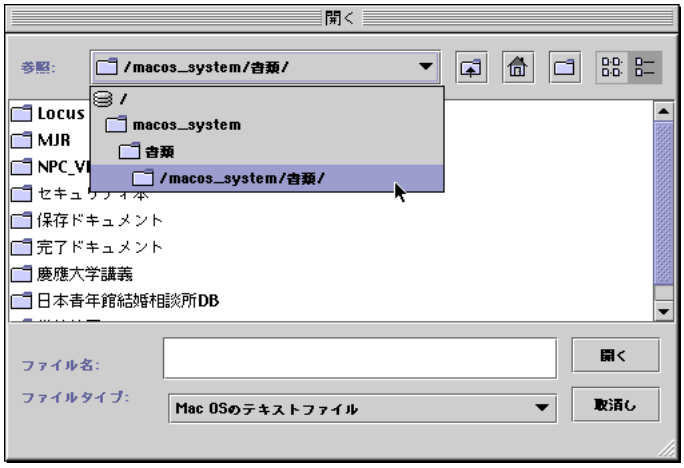

図 8-26　ファイル選択ダイアログボックスでの上位フォルダへの移動コントロール

Swing のファイル処理では、プラットフォームに依存するようなファイル処理も可能なように、javax.swing.filechooser.FileSystemView という抽象クラスが定義されています。実際には、このクラスを継承したきちんと働くクラスをインプリメントして置く必要があるのでしょうけれども、どうも完全には機能しません。このクラスには、ボリュームのルートへの参照を得られたり、隠しファイルかどうかの判断ができるメソッドが定義されているので、java.io.File クラスの機能に足りない面は補完できるはずなのです。

FileSytemView は、スタティックメソッドの getFileSystemView でインスタンスを取得します。ところが、isHiddenFile メソッドはすべて false を戻します。また、ボリュームの一覧を取ることもできますが、起動ボリュームかどうかの判断ができないため、Mac OS では使いづらい機能です。ドライブ番号が固定の Wintel マシン向けの機能のような気がします。

FileChooser では、フィルタ関数を定義してやることで、たとえばテキストファイルだけをダイアログボックスに表示するということもできるのですが、この章のサンプルでは、いろいろ制約が出てきてしまいました。その部分はどうしても JDirect の

機能に頼ることになってしまいます。

さらに、Swing のコンポーネントが戻すファイル名の文字列を、たとえば File などの生成で利用したときなどに、Mac OS のエラーを戻すときがあります。システムの隠しファイルなどで、実際に表示すると化け文字になるようなキャラクタが入っているときにこの現象が発生します。

♦Swing への雑感

機能豊富なコンポーネント群という意味では大いに期待できるのは言うまでもありません。実用的に利用できれば、アプリケーションは非常に作りやすくなります。

一方、現状の Swing が実用的かどうかは一概には言えないでしょう。むしろ、このような状態では、まだ実用的でないと判断される場面が出てくるのではないかと思います。1 つの理由は処理速度の遅さです。ファイル選択をネットワークの先のボリュームで行うと恐ろしく待たされます。Java の弱点だった処理の遅さも次第に解決されてきた矢先、Swing を使うとなると昔に戻されたような気がします。もちろん、近い将来はもっと高速なパソコンが一般的になるのだということを目論んでいるのであれば、必ずしもデメリットにはなりませんが、現状ではかなり高速な機種でないと実用的には使えないと思われます。

また、ファイル選択ダイアログボックスに見られるように、完成度が高くない面もあります。こうした部分を回避しながらプログラムを作るとなると、かなり大変になってきて、「手軽さ」というメリットが削がれると言えるでしょう。

Swing は非常に便利なコンポーネントですが、すべて Swing でそろえずに、場合によっては AWT のものを使うというのが、1 つの妥協点ではないかと思われます。

8-3 サンプルプログラム

Swing を利用したサンプルプログラムのソースを掲載します。テキストビューア、HTML ビューアの機能を持つようなアプリケーションを作成しました。また、BNDL リソースを与えて、ドラッグ&ドロップなどでもファイルを開くようにしています。

実行するための条件

サンプルアプリケーションは、5 つのソースファイルから構成されます。第 4 章の CodeWarrior でのアプリケーション作成方法に従って、プロジェクトを構築します。既定値からの違いは、まず、Mac OS Java Linker をポストリンカとして設定すること、MRJClasses.zip をプロジェクトに加えることは、まずはアプリケーションの基本です。また、BNDL リソースを含むファイルをプロジェクトに追加します。そして、ファイルタイプ rsrc に対して JarImporter を使うように設定を変更し、リソースファイルのインスペクタを開いて、リソースを生成ファイルにマージするようにしておきます。main メソッドがあるのは Draggable というクラスです。以上のような、アプリケーションとしての基本的な設計を行っておきます。

さらに、swingall.jar あるいは、swing.jar とルック&フィールのファイルなど、必要な Swing のコンポーネントをプロジェクトに追加しておきます。

サンプルプログラムは、第 5 章のプログラムをもとに Swing 対応しましたが、ビューアの種類が 2 種類になっているので、ビューアの元になるクラスと、実際のビューアのクラスに別れて、共通の処理を元になるクラスで記述するようにしています。

なお、サンプルプログラムのファイルはダウンロードできるようにしておき、一連のファイルにプロジェクトファイルは作成しておきますが、Swing のライブラリファイルは含めていません。Swing は独自に入手して、プロジェクトのフォルダに追加してください。

Draggable.java

このクラスに main メソッドを配置していますが、アプリケーションの基礎になる部分を定義しています。必須の AppleEvent である OpenDocument などに対応する処理

をここで組み込んでいます。また、ウインドウが何も表示されていないときにメニューが表示されるように、画面外にウインドウを表示して AWT の機能を使ってメニューを作成しています。

```
import com.apple.mrj.*;
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Draggable extends Frame        //Mac OS のアプリケーションの基本となるクラス。
                                            //      Frame を継承した
    implements      MRJOpenDocumentHandler,     //イベントハンドラのインプリメント
                    MRJPrintDocumentHandler,
                    MRJQuitHandler,
                    MRJAboutHandler
{
    static public void main(String arg[])      //起動時に実行されるメソッド
    {
        new Draggable();        //このクラスを新たに生成する
    }

    public Draggable()      //コンストラクタ
    {
        /*基本 AppleEvent が、このクラスに伝達されるように定義する*/
        MRJApplicationUtils.registerOpenDocumentHandler(this);      //イベントハンドラ
        MRJApplicationUtils.registerPrintDocumentHandler(this);     //への登録
        MRJApplicationUtils.registerQuitHandler(this);
        MRJApplicationUtils.registerAboutHandler(this);

        setupMenuBar(); //メニューバーの設定

        setBounds(-1000,-100,10,10);    //ウインドウの位置を画面の外側に出す。
            //      つまり非表示にしたいのだが、hide()とするとメニューまで消えるので、
            //      こういう方針にした
        show();     //ウインドウを表示するが、実際には画面には見えない
    }

    public void handleOpenFile(File filename)       //Open イベントが Finder からやってきたとき
    {
        System.out.println("Dragged "+filename.toString()); //コンソールへの出力
        ViewerSeed_S.openDocumentByViewer(filename);
                                            //開くファイルを TextViewer_S で表示する
    }

    public void handlePrintFile(File filename)      //Print イベントが Finder からやってきたとき
    {
        System.out.println("Print Event "+filename.toString());     //コンソールへの出力
        ViewerSeed_S newViewer = ViewerSeed_S.openDocumentByViewer(filename);
                                            //開くファイルを TextViewer_S で表示する
        newViewer.printDocumentByViewer();  //開いたビューアで印刷を行う
    }
```

```
//Quit イベントが Finder からやってきたとき、
//      あるいはアップルメニューの「終了」を選択したとき
public void handleQuit()
{
        System.out.println("Quit Event Received");     //コンソールへの出力
        System.exit(0);                                 //アプリケーションの終了
}

public void handleAbout()     //アップルメニューの About を選択したとき
{
        System.out.println("Select About Menu");       //コンソールへの出力
}

/*      TextViewer_S のウインドウが何も表示されていないときにもメニューを表示される
        ようにしたいそのような場合に必要なメニューに絞って表示するが、メニューを
        表示するにはそもそも Frame でなければならない。このクラスは Frame を拡張した
        が、そのウインドウは表示したくないので、コンストラクタで画面外に追いやった
*/
private void setupMenuBar()
{
        MenuItem newItem, openItem, closeItem, quitItem;

        MenuBar mb = new MenuBar();    //メニューバーを新たに生成

        Menu fileMenu = new Menu("ファイル", true);        //「ファイル」メニューを作成
        newItem = new MenuItem("新規", new MenuShortcut('N'));
                                        //「新規」の項目を生成し、ショートカットは N
        fileMenu.add(newItem);       //項目をメニューに追加する
        openItem = new MenuItem("開く...", new MenuShortcut('O'));
                                        //「開く」の項目を生成し、ショートカットは O
        fileMenu.add(openItem);      //項目をメニューに追加する
        fileMenu.addSeparator();     //区切り線を入れる
        quitItem = new MenuItem("終了", new MenuShortcut('Q'));
                                        //「終了」の項目を生成し、ショートカットは Q
        fileMenu.add(quitItem);      //項目をメニューに追加する
        mb.add(fileMenu);            //メニューをメニューバーに追加する

        NewItemListener newListener = new NewItemListener();
                                        //「新規」の処理クラスを生成する
        newItem.addActionListener(newListener);
                                        //メニュー項目を選択すると呼び出されるようにする

        OpenItemListener openListener = new OpenItemListener();
                                        //「開く」の処理クラスを生成する
        openItem.addActionListener(openListener);
                                        //メニュー項目を選択すると呼び出されるようにする

        QuitItemListener quitListener = new QuitItemListener();
                                        //「終了」の処理クラスを生成する
        quitItem.addActionListener(quitListener);
                                        //メニュー項目を選択すると呼び出されるようにする

        setMenuBar(mb);       //メニューバーをウインドウに登録する
```

```
    }

    class NewItemListener implements ActionListener  {
        public void actionPerformed(ActionEvent ev)  {      //「新規」で呼び出される
            ViewerSeed_S.newDocumentByViewer();      //ウインドウを新たに作成する
        }
    }

    class OpenItemListener implements ActionListener {
        public void actionPerformed(ActionEvent ev)  {      //「開く」で呼び出される
            ViewerSeed_S.openDocByViewerWithDialog(Draggable.this);
                //ダイアログを表示してファイルを指定し、そのファイルを開く
        }
    }

    class QuitItemListener implements ActionListener  {
        public void actionPerformed(ActionEvent ev)  {      //「終了」で呼び出される
            System.exit(0);  //アプリケーションを終了する
        }
    }
}
```

ViewerSeed_S.java

ビューアの元になる抽象クラスです。まず、ビューアはいずれもメニューを持っていたり、ツールバーを持っている点は共通です。ここでは、ビューアごとにメニューを変えるようなことも行っていないので、メニュー構築やツールバー構築を、このクラスで最初から行っています。また、メニューやボタンクリックなどのイベントの受付も内部クラスで行っています。ただし、実際の処理はビューアに使うコンポーネントに依存するので、abstract で定義したメソッドを呼び出すことにし、実際の処理はビューアごとに作成するようにしています。

また、インスタンス化するときに、使用するルック&フィールを設定しています。プログラム的には、単に設定するルック&フィールの命令をコメントからはずすようにしているだけですので、適当にコメントをつけたりはずしたりして、任意のルック&フィールを使用して下さい。

このクラスの中には、新しくビューアのウインドウを表示するのと、指定したファイルをビューアで開くメソッドを用意していますが、それらは static にしています。ファイルを開くメソッド内では、拡張子が html ないしは htm だと HTML ビューア、そうでないなら JTextArea のビューアを利用するようにしています。こうした機能は開いているビューアから呼び出されるだけでなく、OpenDocument イベントがあった

ときに Draggable クラスから呼び出されるなど、いろいろな利用パターンがあります。

そこで、static にして、汎用ルーチン的な定義にしてみました。

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import javax.swing.*;
import com.apple.mrj.*;

/*  ビューアのもとになるクラス。Swing 対応で、JFrame を継承している    */
abstract class ViewerSeed_S extends JFrame
{
    File openFile = null;           //開いているファイル
    Dimension initilaWindowSize = new Dimension(500,300);       //ウインドウの初期サイズ
    boolean isDirty = false;        //修正したかどうかを記録するフラグ

    public ViewerSeed_S()       //デフォルトのコンストラクタ
    {
        setuPLookAndFeel();  //ルック&フィールを設定する
        setupMenuBar();         //メニューを構築する
        setupToolBar();         //ツールバーを構築する
    }

    private void setuPLookAndFeel()
    {
        System.out.println(
            "System Look & Feel: "+ UIManager.getSystemLookAndFeelClassName());
                //システムの既定のルック&フィールをコンソールに出力
        //ルック&フィールを設定する。必要な行だけを生かして、あとはコメントにしておく
        try    {
//          UIManager.setLookAndFeel("com.sun.java.swing.plaf.motif.MotifLookAndFeel");
//          UIManager.setLookAndFeel("com.sun.java.swing.plaf.metal.MetalLookAndFeel");
                                                                //これは使えない
//          UIManager.setLookAndFeel("javax.swing.plaf.metal.MetalLookAndFeel");
                                                                //既定のルック&フィール
//          UIManager.setLookAndFeel("com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
//          UIManager.setLookAndFeel("com.sun.java.swing.plaf.mac.MacLookAndFeel");
                            //これを使うにはライブラリに mac.jar が必要
//          UIManager.setLookAndFeel("com.sun.java.swing.plaf.macos.MacOSLookAndFeel");
                            //これを使うにはライブラリに macos.jar が必要
        }           catch(Exception e)      {
                    System.out.println("...Use default Look and Feel");
                    System.out.println("...Missing..." + e.getMessage());
        }
    }

    private JMenuItem saveItem;     //「保存」の項目は状況によって選択できなくするため、
                                    //別のメソッドから利用できるようにメンバ変数にしておく

    private void setupMenuBar() //メニューを構築する
    {
        JMenuItem newItem, openItem, saveAsItem, closeItem, printItem, quitItem;
                                                                //「ファイル」メニューの項目
```

```
JMenuItem undoItem, cutItem, copyItem, pasteItem;       //「編集」メニューの項目
JMenuItem bgWhite, bgGray, bgYellow;
                                        //「編集」メニューの「背景色」のサブメニュー項目
JMenuItem aboutMRJ;                     //「ヘルプ」メニューに追加する項目

JMenuBar mb = new JMenuBar();           //メニューバーを用意する

/*「ファイル」メニューの作成*/
JMenu fileJMenu = new JMenu("ファイル", true);    //「ファイル」メニューを作成する
//      fileJMenu.setMnemonic(KeyEvent.VK_F);
                                        //ファイルメニューにニーモニックを指定する場合

newItem = new JMenuItem("New", KeyEvent.VK_N);       //「新規」項目を作成
newItem.setAccelerator(
        KeyStroke.getKeyStroke(KeyEvent.VK_N, ActionEvent. META_MASK ));
                                        //アクセラレーターキーは N
fileJMenu.add(newItem);                 //「ファイル」メニューに追加する
NewItemListener newListener = new NewItemListener();
        //「新規」を選択したときにイベントを受け付けるクラスを新たに生成
newItem.addActionListener(newListener);
        //「新規」を選んだときにイベントが発生するように設定

openItem = new JMenuItem("開く(O)...", KeyEvent.VK_O);      //「開く」項目を作成
openItem.setAccelerator(
        KeyStroke.getKeyStroke(KeyEvent.VK_O, ActionEvent. META_MASK ));
                                        //アクセラレーターキーは O
fileJMenu.add(openItem);                //「ファイル」メニューに追加する
OpenItemListener openListener = new OpenItemListener();
        //「開く」を選択したときにイベントを受け付けるクラスを新たに生成
openItem.addActionListener(openListener);
        //「開く」を選んだときにイベントが発生するように設定

closeItem = new JMenuItem("閉じる", KeyEvent.VK_W); //「閉じる」項目を作成
closeItem.setAccelerator(
        KeyStroke.getKeyStroke(KeyEvent.VK_W, ActionEvent. ALT_MASK ));
                                        //アクセラレーターキーは W
fileJMenu.add(closeItem);               //「ファイル」メニューに追加する
CloseItemListener closeListener = new CloseItemListener();
        //「閉じる」を選択したときにイベントを受け付けるクラスを新たに生成
closeItem.addActionListener(closeListener);
        //「閉じる」を選んだときにイベントが発生するように設定

fileJMenu.addSeparator();               //区切り線を追加する

saveItem = new JMenuItem("保存", KeyEvent.VK_S);      //「保存」項目を作成
saveItem.setAccelerator(
        KeyStroke.getKeyStroke(KeyEvent.VK_S, ActionEvent. META_MASK ));
                                        //アクセラレーターキーは S
saveItem.setEnabled(false);             //「保存」を選択できないようにイネーブルでなくす
fileJMenu.add(saveItem);                //「ファイル」メニューに追加する
saveAsItem = new JMenuItem("名前を付けて保存...");
                                //「名前を付けて保存」項目を作成、アクセラレーターキーはなし
fileJMenu.add(saveAsItem);              //「ファイル」メニューに追加する
SaveItemListener saveListener = new SaveItemListener();
```

```
            //「保存」「名前を付けて保存」を選択したときに
            //イベントを受け付けるクラスを新たに生成
saveItem.addActionListener(saveListener);
            //「保存」を選んだときにイベントが発生するように設定
saveAsItem.addActionListener(saveListener);
            //「名前を付けて保存」を選んだときにイベントが発生するように設定

fileJMenu.addSeparator();           //区切り線を追加する

printItem = new JMenuItem("印刷", KeyEvent.VK_P);      //「印刷」項目を作成
printItem.setAccelerator(
            KeyStroke.getKeyStroke(KeyEvent.VK_P, ActionEvent. META_MASK ));
                                    //アクセラレーターキーは P
fileJMenu.add(printItem);           //「ファイル」メニューに追加する
PrintItemListener printListener = new PrintItemListener();
            //「印刷」を選択したときにイベントを受け付けるクラスを新たに生成
printItem.addActionListener(printListener);
            //「印刷」を選んだときにイベントが発生するように設定

fileJMenu.addSeparator();           //区切り線を追加する

quitItem = new JMenuItem("終了", KeyEvent.VK_Q);       //「終了」項目を作成
quitItem.setAccelerator(
            KeyStroke.getKeyStroke(KeyEvent.VK_Q, ActionEvent. META_MASK ));
                                    //アクセラレーターキーは Q
fileJMenu.add(quitItem);            //「ファイル」メニューに追加する
QuitItemListener quitListener = new QuitItemListener();
            //「終了」を選択したときにイベントを受け付けるクラスを新たに生成
quitItem.addActionListener(quitListener);
            //「終了」を選んだときにイベントが発生するように設定

mb.add(fileJMenu);              //「ファイル」メニューをメニューバーに追加する

/*「編集」メニューの作成*/
JMenu editJMenu = new JMenu("編集", true); //「編集」メニューを作成する
EditItemListener editListener = new EditItemListener();
            //「編集」メニューを選択したときの処理を行うクラスを新たに生成する

undoItem = new JMenuItem("やり直し", KeyEvent.VK_Z);       //「やり直し」項目を作成
undoItem.setAccelerator(
            KeyStroke.getKeyStroke(KeyEvent.VK_Z, ActionEvent. META_MASK ));
                //アクセラレーターキーは Z
undoItem.setEnabled(false);
                //「やり直し」を選択できないようにイネーブルでなくす
editJMenu.add(undoItem);            //「編集」メニューに追加する
undoItem.addActionListener(editListener);
                //「やり直し」を選択したときにイベントが発生するようにしておく

editJMenu.addSeparator();           //区切り線を追加する

cutItem = new JMenuItem("カット", KeyEvent.VK_X);     //「カット」項目を作成
cutItem.setAccelerator(
            KeyStroke.getKeyStroke(KeyEvent.VK_X, ActionEvent. META_MASK ));
                                    //アクセラレーターキーは X
```

```
        editJMenu.add(cutItem);             //「編集」メニューに追加する
        cutItem.addActionListener(editListener);
                //「カット」を選択したときにイベントが発生するようにしておく

        copyItem = new JMenuItem("コピー", KeyEvent.VK_C);  //「コピー」項目を作成
        copyItem.setAccelerator(
                KeyStroke.getKeyStroke(KeyEvent.VK_C, ActionEvent. META_MASK ));
                                        //アクセラレーターキーは C
        editJMenu.add(copyItem);            //「編集」メニューに追加する
        copyItem.addActionListener(editListener);
                //「コピー」を選択したときにイベントが発生するようにしておく

        pasteItem = new JMenuItem("ペースト", KeyEvent.VK_V);       //「ペースト」項目を作成
        pasteItem.setAccelerator(
                KeyStroke.getKeyStroke(KeyEvent.VK_V, ActionEvent. META_MASK ));
                                        //アクセラレーターキーは V
        editJMenu.add(pasteItem);           //「編集」メニューに追加する
        pasteItem.addActionListener(editListener);
                //「ペースト」を選択したときにイベントが発生するようにしておく

        editJMenu.addSeparator();           //区切り線を追加する

        /*「編集」メニュー「背景色」でのサブメニューの作成*/
        JMenu bgColorJMenu = new JMenu("背景色");
                                            //サブメニューの元になる「背景色」項目を作成
        bgWhite = new JMenuItem("白");          //「白」項目を作成
        bgColorJMenu.add(bgWhite);              //「背景色」のサブメニューに追加する
        bgGray = new JMenuItem("グレイ");       //「グレイ」項目を作成
        bgColorJMenu.add(bgGray);               //「背景色」のサブメニューに追加する
        bgYellow = new JMenuItem("黄色");       //「黄色」項目を作成
        bgColorJMenu.add(bgYellow);             //「背景色」のサブメニューに追加する
        editJMenu.add(bgColorJMenu);
                                    //「背景色」のサブメニューを「編集」メニューに追加する

        BackgroundColorItemListener bgListener = new BackgroundColorItemListener();
            //「背景色」のサブメニューを選択したときの処理を行うクラスを新たに生成する
        bgWhite.addActionListener(bgListener);
            //「白」を選択したときにイベントが発生するようにしておく
        bgGray.addActionListener(bgListener);
            //「グレイ」を選択したときにイベントが発生するようにしておく
        bgYellow.addActionListener(bgListener);
            //「黄色」を選択したときにイベントが発生するようにしておく

        mb.add(editJMenu);    //「編集」メニューをメニューバーに追加する

        setJMenuBar(mb);      //メニューバーを Frame に設定する

        /*「ヘルプ」メニューへの追加*/
        JMenu helpJMenu = new JMenu("ヘルプ");    //「ヘルプ」メニューを作成する
        helpJMenu.add(aboutMRJ = new JMenuItem("MJR とは？"));
                                            //「MJR とは？」という項目を追加する
        mb.add(helpJMenu);                  //「ヘルプ」メニューをメニューバーに追加する
//      mb.setHelpMenu(helpJMenu);      //ここで作成したメニューをヘルプメニューとする
            /*      これにより、システムが用意する「ヘルプ」メニューと一体化する
```

```
                ただし、Swing1.1.1beta のドキュメントでは、このメソッドは組み込まれ
                ていないとなっている。
                メソッドがあればエラーでストップするようなので、とりあえずコメント
                化しておく                                */

        aboutMRJ.addActionListener(
                //ヘルプメニューに追加した項目を選択したときの処理を直接記述している
            new ActionListener()        {
                public void actionPerformed(ActionEvent ev)  {
                    System.out.println("Help");
                                //処理は単にコンソールに文字を出力するだけ
                }     }     );
    }

    private void setupToolBar()  //ツールバーの設定
    {
        JToolBar tb = new JToolBar();     //ツールバーを新しく作成する

        JButton newButton = new JButton("新規", new ImageIcon("icons/new.gif"));
                                        //ボタンを生成する。ボタン名は「新規」で、
                                        //指定した GIF ファイルのアイコンをつける
        newButton.setToolTipText("新しくウインドウを開く");   //チップヒントの文句
        tb.add(newButton);    //ボタンをツールバーに登録する
        NewItemListener newListener = new NewItemListener();
                //「新規」をクリックしたときにイベントを受け付けるクラスを新たに生成
        newButton.addActionListener(newListener);
                //「新規」をクリックしたときにイベントが発生するように設定

        JButton openButton = new JButton("開く", new ImageIcon("icons/open.gif"));
                                        //ボタンを生成する。ボタン名は「開く」で、
                                        //指定した GIF ファイルのアイコンをつける
        openButton.setToolTipText("ファイルを新しいウインドウで開く");
                                        //チップヒントの文句
        tb.add(openButton);   //ボタンをツールバーに登録する
        OpenItemListener openListener = new OpenItemListener();
                //「開く」をクリックしたときにイベントを受け付けるクラスを新たに生成
        openButton.addActionListener(openListener);
                //「開く」をクリックしたときにイベントが発生するように設定

        this.getContentPane().add(tb, BorderLayout.NORTH);
                //ツールバーをウインドウに登録する。
                //BorderLayout の機能を利用し、ウインドウの上端に配置する
    }

/*「ファイル」メニューの「新規」を選択、あるいは「新規」ボタンをクリックしたときの処理*/
    class NewItemListener implements ActionListener  {
                public void actionPerformed(ActionEvent ev)     {
            newDocumentByViewer();          //新規ドキュメントを用意する
        }
    }

/*「ファイル」メニューの「開く」を選択、あるいは「開く」ボタンをクリックしたときの処理*/
    class OpenItemListener implements ActionListener {
        public void actionPerformed(ActionEvent ev)  {
```

```
                openDocByViewerWithDialog(ViewerSeed_S.this);
                        //開くファイルをダイアログボックスで選択して実際に開く
        }
}

/*      実際に新規ドキュメントを用意する。
        ウインドウ外からも利用できるように static にしてある*/
static void newDocumentByViewer()       {
        new TextViewer_S(null);       //TextViewer_S のウインドウを新たに作成する
}

/*      ファイル選択ダイアログボックスを表示してファイルを選び、ビューアで開く。
        ウインドウ外からも利用できるように static にしてある*/
static void openDocByViewerWithDialog(Component parentWindow)          {
        File rootFolder = null;
        try     {
                rootFolder = MRJFileUtils.findFolder(new MRJOSType("docs"));
                                                                //「書類」フォルダへの参照を得る
        }               catch(Exception e)      {System.out.println(e.getMessage());            }
        JFileChooser fc = new JFileChooser(rootFolder);
                //初期フォルダを「書類」フォルダにしたファイル選択ダイアログボックスを作成
        fc.setFileFilter(new MacTextFileFilter());
                //テキストファイルだけを表示するフィルタを適用する
        int returnValue = fc.showOpenDialog(parentWindow);      //ダイアログボックスを開く
        if (returnValue == JFileChooser.APPROVE_OPTION)      //ファイルを選択したのなら
                openDocumentByViewer(fc.getSelectedFile());       //そのファイルを実際に開く
}

/*      指定したファイルを開くが拡張子に応じて利用するビューアを選択し、
        実際にそのビューアで開く。
        ウインドウ外からも利用できるように static にしてある*/
static ViewerSeed_S openDocumentByViewer(File targetFile)   {
        ViewerSeed_S instancedViewer = null;
        String fileNameLower = targetFile.getName().toLowerCase();
                //開くファイルのファイル名の小文字文字列を得る
        if(fileNameLower.endsWith(".htm") || fileNameLower.endsWith(".html"))
                instancedViewer = new HTMLViewer_S(targetFile);
                        //ファイルの末尾が.htm あるいは.html なら、HTML ビューアを用意する
        else
                instancedViewer = new TextViewer_S(targetFile);
                        //そうでなければ、テキストビューアを用意する
        return instancedViewer;       //生成したオブジェクトへの参照を戻す
}

/*「ファイル」メニューの「閉じる」を選択したときの処理*/
class CloseItemListener implements ActionListener {
        public void actionPerformed(ActionEvent ev)   {
                /*本来は修正されていれば保存する処理が必要*/
                dispose();          //Frame を閉じる
        }
}

/*「ファイル」メニューの「保存」「名前を付けて保存」を選択したときの処理*/
class SaveItemListener implements ActionListener  {
```

```
        public void actionPerformed(ActionEvent ev)  {
            String itemLabel = ((JMenuItem)ev.getSource()).getLabel();
                                                    //選択したメニューの項目名を取り出す
            if((openFile == null) || (itemLabel.compareTo("名前を付けて保存...") == 0))
    {
                                                    //選択したのが「名前を付けて保存...」なら
                File rootFolder = null;
                try    {
                    rootFolder = MRJFileUtils.findFolder(new MRJOSType("docs"));
                                                        //「書類」フォルダへの参照を得る
                }           catch(Exception e)     {System.out.println(e.getMessage());
    }
                JFileChooser fc = new JFileChooser(rootFolder);
                    //「書類」フォルダを初期フォルダにしたファイル選択
                    //      ダイアログボックスを生成
                int returnValue = fc.showSaveDialog(ViewerSeed_S.this);
                    //保存するファイルを指定するダイアログボックスを開く
                if (returnValue == JFileChooser.APPROVE_OPTION)     {           //保存する
なら
                    openFile = fc.getSelectedFile();    //保存するファイルを記録する
                    setTitle(openFile.getName());
                                //ファイル名をウインドウのタイトルにする
                    setDirty();  //変更したことを示すフラグを強制的に設定する
                }
                else
                    return;
            }
            if (isDirty)  {      //表示している中身に修正があれば
                doSave();          //それをファイルに保存する
                resetDirty();      //修正したことを示すフラグをクリアする
            }
        }
    }

    abstract void doSave();       //実際の保存処理は、継承したクラスで記述する

    /*「ファイル」メニュー印刷」を選択したときの処理*/
    class PrintItemListener implements ActionListener  {
        public void actionPerformed(ActionEvent ev)  {
            printDocumentByViewer();   //実際に印刷を行う
        }
    }

    /*     実際に印刷を行う。ウインドウ外からも利用できるように static にしてある*/
    void printDocumentByViewer()     {
        PrintJob pj = this.getToolkit().getPrintJob(this,"A",null);
                                //印刷設定のダイアログボックスが表示される
        Graphics pg = pj.getGraphics();    //印刷時のグラフィックス環境を取得
        if(pg != null)      {
            doPrint(pg);            //印刷メソッドを呼び出す
            pg.dispose();           //グラフィックス領域の破棄により、実際に印刷が始まる
        }
        pj.end();
    }
```

```
abstract void doPrint(Graphics pg);       //具体的な印刷作業は、継承したクラスで記述する

/*「ファイル」メニューの「終了」を選択したときの処理*/
class QuitItemListener implements ActionListener   {
        public void actionPerformed(ActionEvent ev)  {
                System.exit(0);   //アプリケーションを終了
        }
}

/*「編集」メニューを選択したときの処理*/
class EditItemListener implements ActionListener   {
        public void actionPerformed(ActionEvent ev)  {
                String itemLabel = ((JMenuItem)ev.getSource()).getLabel();
                                                        //選択したメニューの項目名を取り出す
                if(itemLabel.compareTo("やり直し") == 0)      //選択したのが「やり直し」なら
                        doUndo();  //実際の処理を行うメソッドを呼び出す
                else if(itemLabel.compareTo("カット") == 0)  //選択したのが「カット」なら
                        doCut();     //実際の処理を行うメソッドを呼び出す
                else if(itemLabel.compareTo("コピー") == 0)  //選択したのが「コピー」なら
                        doCopy();  //実際の処理を行うメソッドを呼び出す
                else if(itemLabel.compareTo("ペースト") == 0)//選択したのが「ペースト」なら
                         doPaste(); //実際の処理を行うメソッドを呼び出す
        }
}

abstract void doUndo();       //実際のやりなおし処理は、継承したクラスで記述する
abstract void doCut();        //実際のカットの処理は、継承したクラスで記述する
abstract void doCopy();       //実際のコピー処理は、継承したクラスで記述する
abstract void doPaste();      //実際のペースト処理は、継承したクラスで記述する

/*「編集」メニューの「背景色」から項目をを選択したときの処理*/
class BackgroundColorItemListener implements ActionListener        {
        public void actionPerformed(ActionEvent ev)  {
                String itemLabel = ((JMenuItem)ev.getSource()).getLabel();
                                                        //選択したメニューの項目名を取り出す
                if(itemLabel.compareTo("白") == 0)              //選択したのが「白」なら
                        setPaneBackground(Color.white);        //背景を白にする
                else if(itemLabel.compareTo("グレイ") == 0)  //選択したのが「グレイ」なら
                        setPaneBackground(Color.gray);         //背景をグレイにする
                else if(itemLabel.compareTo("黄色") == 0)     //選択したのが「黄色」なら
                        setPaneBackground(Color.yellow);       //背景を黄色にする
        }
}

abstract void setPaneBackground(Color c);
        //内部の色を設定する処理は、継承したクラスで記述する

/*      修正したことを記録し、「保存」メニューを使えるようにする。*/
public void setDirty()   {
        isDirty = true;
        saveItem.setEnabled(true);  //「保存」を選択できるようにする
}
```

```
    /*      修正した記録をクリアし、「保存」メニューは使えなくする*/
    public void resetDirty() {
        isDirty = false;
        saveItem.setEnabled(false); //「保存」を選択できないようにする
    }
}
```

MacTextFileFilter.java

JFileChooser でファイル選択のダイアログボックスを表示するときに、ここで定義した MacTextFileFilter をフィルタ関数として指定することで、一覧にはフォルダとテキストファイルだけが表示されるようにしています。同様な機能を第 6 章で、AWT の FileDialog 向けのフィルタクラスとして紹介していますが、その段階では、MRJ 2.1EA3 が利用できるバージョンでした。そのバージョンで、非表示ファイルかどうかをチェックする機能が、MRJ 2.1 正式版以降では使えなくなっており、別の方法に変わっています。ここでは、正式版での手法を利用します。ここで、ファイルの Finder 情報を取得する方法は分かったのですが、フォルダの情報を取得する方法が判明しませんでした。ファイルについては非表示ファイルはダイアログボックスには表示されないのですが、フォルダについては非表示のものも表示されてしまいます。

```
import java.io.*;
import javax.swing.filechooser.*;
import com.apple.mrj.MRJOSType;
import com.apple.mrj.MRJFileUtils;
import com.apple.MacOS.FSSpec;
import com.apple.MacOS.FInfo;          //MRJ2.1 正式版で初登場のクラス！？
//import com.apple.mrj.macos.generated.FinderConstants;     //MRJ2.1 正式版ではなくなった…

/*Swing の FileChooser で、テキストファイルとフォルダだけを表示するためのフィルタクラス*/
class MacTextFileFilter extends FileFilter    {
    public boolean accept(File f) {       //引数 f のファイルを表示するなら true を戻すメソッド
        boolean     returnValue = false;
        MRJOSType fType = null;

        if(f.isDirectory())  //フォルダは無条件にリストに含める
            /*      そのため、非表示フォルダも表示される。フォルダが非表示かどうかを
                求める機能が見つからない…        */
            returnValue = true;
        else  {       //以降、ファイルの場合
            try     {
                fType = MRJFileUtils.getFileType(f);      //ファイルタイプを取得する
                    /*デスクトップ管理の非表示ファイルなど、化け文字のあるファイル
                    では例外が発生してしまうものの、処理はストップしない。その意味
                    では、それらのファイルは自動的にリストに含めない。このエラーは、
                    化ける文字に関して Swing で得られたファイル文字列と、MRJ での
```

```
                        ファイル文字列が一致していないために起こると思われる。*/
                } catch(Exception e)    {
                        System.out.println(e.getMessage());
                        return false;       }
                            //例外が発生するファイルはともかくリストに含めない
                if(fType.equals(new MRJOSType("TEXT")))   //ファイルタイプが TEXT なら
                    returnValue = isNotVisible(f);
                            //非表示フラグがセットされていないならでリストに含める
                else  //ファイルタイプが TEXT でないなら、リストに含めない
                    returnValue = false;
            }
            return returnValue;
        }
        public String getDescription()      {
            return("Mac OS のテキストファイル");
        }

        /*  非表示フラグがセットされているかを調べる。されていなければ true を戻す。
            このメソッドはファイルにしか利用できない。
            また、化け文字が含まれるようなファイル（システムが作る隠しファイル）において、
            FInfo のコンストラクタでエラーが出る模様。例外ではなく、処理がストップしてしまう。
            たぶん、ファイル名の文字列がうまく処理されていないのだろうと思われる。
                99/3/21 MRJ2.1, Swing 1.1.1beta1*/

        private boolean isNotVisible(File targetFile)    {
            try{
                FSSpec targetSpec = new FSSpec(targetFile);
                                                    //FSSpec オブジェクトを構築する
                FInfo finderInfos = new FInfo(targetSpec);     //ファイルの Finder 情報を取得する
                if ((finderInfos.getFlags() & 0x4000) != 0)
                    //非表示のフラグをチェック。定数が定義されているクラスが見当たらない…
                    return false;       //フラグがセットされている
                else
                    return true; //非表示フラグがセットされていない
            }   catch(Exception e)
            {           return false;       }

    /*  //Swing の FileSystemView で非表示項目のチェックができるはずだが、機能していない。
        すべてのファイルで false を戻す
            FileSystemView fsv = FileSystemView.getFileSystemView();
            return(!fsv.isHiddenFile(targetFile));
        */
        }
    }
```

TextViewer_S.java

JTextArea を利用したテキストビューアのクラスです。メニューやツールバーの構築、アクションイベントの受付などは、ViewerSeed_S クラスで定義してしまっている

ので、このクラスでやっていることの半分以上は、JScrollPane と JTextArea を配置し、テキストファイルから読み出すところです。また、ファイルへの書き出しも作成しておきましたが、ファイルに書き出した後は、ファイルタイプの設定も行っています。

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import javax.swing.*;
import com.apple.mrj.*;

/*   ViewerSeed_S クラスを継承して、テキストビューアを定義する*/
public class TextViewer_S extends ViewerSeed_S
{
    private JTextArea viewText; //ウインドウ内に配置する TextArea

    public TextViewer_S(File targetFile)
    {
        openFile = targetFile;  //引数のファイルをメンバ変数に保存

        /*テキストを表示するための JTextArea と JScrollPane の用意*/
        viewText = new JTextArea();        //JTextArea を新たに用意する
        viewText.setLineWrap(true);
                                    //行の折り返しを行うようにする（動作は遅くなるが…）
        viewText.setWrapStyleWord(true);        //ワード単位の折り返しを行う
        viewText.addKeyListener(new textAreaKeyListener());     //キーイベントに反応させる
        JScrollPane scrollingPane = new JScrollPane(viewText);          //スクロール領域を用意
        scrollingPane.setPreferredSize(initilaWindowSize);
                                                    //スクロール領域の要求サイズを指定する
        scrollingPane.setHorizontalScrollBarPolicy(
            JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);
                                                    //水平スクロールバーを表示しない

        Container windowContents = getContentPane();
                                        //ウインドウの中身を管理するオブジェクトを取得
        windowContents.add(scrollingPane, BorderLayout.CENTER);
                                        //TextArea をウインドウに追加する

        /*ファイルから読み込んだ内容を TextArea に設定する*/
        if (openFile != null)     {       //ファイルが指定されているかをチェック
            setTitle(openFile.getName());       //ウインドウのタイトルはファイル名にする
            try    {
                FileReader LNR = new FileReader(targetFile);
                    //引数に指定したファイルから行読み込みする Reader を生成
                viewText.read(LNR, null);
                    //ファイルから読み取ったテキストを JTextArea に設定
                LNR.close();       //ファイルを閉じる
            }
            catch(Exception e)     {       //例外があったときの処理
                System.out.println(e.getMessage());}//エラーメッセージをコンソールに表示
        }
        pack();      //包含するオブジェクトのサイズにウインドウサイズを合わせる
```

```
        show();     //ウインドウを表示する
}

/*      キーを押したときに発生するイベントを処理するクラスを定義       */
class textAreaKeyListener implements KeyListener {
        public void keyTyped(KeyEvent e){       //キーが押されたとき
                setDirty();  //表示内容を修正したことにする
        }
        public void keyPressed(KeyEvent e)      {       }
        public void keyReleased(KeyEvent e)     {       }
}

void doUndo()   {       /*実装せず…*/           }
void doCut()    {       viewText.cut();         }       //カットの処理
void doCopy()   {       viewText.copy();        }       //コピーの処理
void doPaste()  {       viewText.paste();       }       //ペーストの処理

void setPaneBackground(Color c) //背景色の設定
{
        viewText.setBackground(c); //背景色を設定し
        viewText.repaint();                     //再描画
}

void doSave()   {       //保存の処理を記述する
        if (openFile != null)     {       //ファイルが指定されているかを一応チェック
                try     {
                        FileWriter fWriter = new FileWriter(openFile);
                                //ファイルへ書き込みをする FileWriter を生成する
                        viewText.write(fWriter);        //JTextArea の内容をファイルに書き込む
                        fWriter.close();    //FileWriter を閉じる
                        MRJFileUtils.setFileTypeAndCreator(
                                openFile, new MRJOSType("TEXT"), new MRJOSType("MJR1"));
                                        //作成したファイルのタイプとクリエイターを設定する
                }
                catch(Exception e)      {       //例外があったときの処理
                        System.out.println(e.getMessage());}//エラーメッセージをコンソールに表示
        }
}

void doPrint(Graphics g)      {       //印刷の処理を記述する
        String targetLine;              //1 行分を保存する変数
        int currentBase = 0;            //ベースラインを計測する変数
        int LinePitch = 16;             //行ピッチを設定
        StringTokenizer eachLines = new StringTokenizer(viewText.getText(), "¥n", false);
                //JTextArea の内容を改行で区切ったトークンに分解する
        while(eachLines.hasMoreTokens())      {       //トークンを順番に取得
                targetLine = eachLines.nextToken();     //現在のトークンを得る
                currentBase += LinePitch;    //ベースラインを行ピッチ分進める
                g.drawString(targetLine, 0, currentBase);       //行を描画する
                        /*      単に描画しているだけで、用紙幅で折り返すようなことはしていない。
                                ページ送りもしていない      */
        }
        /*      以下のメソッド呼び出しだけでも印刷できる。その場合、ウインドウ内の JTextArea
                に表示されているような改行位置で折り返されるなどするが、ページ制御はしてい
```

```
            ないので、テキストすべてが印刷されるわけではない。
        viewText.print(g);
        */
    }
}
```

HTMLViewer_S.java

JEditorPane を利用した HTML ビューアのクラスです。これもテキストのビューアと同様に、ViewerSeed_S クラスを継承して作っています。また、表示だけをすることを意図しているので、コンストラクタで必要なコンポーネントを用意するだけにしてあります。

```
import java.awt.*;
import java.io.*;
import javax.swing.*;
import com.apple.mrj.*;

/*  ViewerSeed_S クラスを継承して、HTML ビューアを定義する*/
public class HTMLViewer_S extends ViewerSeed_S
{
    JEditorPane viewHTML;      //HTML エディタのオブジェクト

    public HTMLViewer_S(File targetFile)
    {
        openFile = targetFile;  //引数のファイルをメンバ変数に保存

        /*テキストを表示するための JEditorPane と JScrollPane の用意*/
        try   {
            viewHTML = new JEditorPane("file://" + targetFile.getPath());
                        //指定したファイルを表示する JEditorPane を新たに用意する
        }           catch(IOException e)  {      System.out.println(e.getMessage());    }
        JScrollPane scrollingPane = new JScrollPane(viewHTML);      //スクロール領域を用意
        scrollingPane.setPreferredSize(initilaWindowSize);
                                                    //スクロール領域の要求サイズを指定する
        scrollingPane.setHorizontalScrollBarPolicy(
            JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);
                                                    //水平スクロールバーを表示しない

        Container windowContents = getContentPane();
                                        //ウインドウの中身を管理するオブジェクトを取得
        windowContents.add(scrollingPane);    //JEditorPane をウインドウに追加する

        setTitle(targetFile.getName());      //ウインドウのタイトルはファイル名にする

        pack();      //包含するオブジェクトのサイズにウインドウサイズを合わせる
        show();      //ウインドウを表示する
    }
```

```
    /*      各種の処理は省略しました…        */
    void doUndo()    {       }
    void doCut()     {       }
    void doCopy()    {       }
    void doPaste()   {       }
    void setPaneBackground(Color c) {       }
    void doSave()    {       }
    void doPrint(Graphics g)     {       }
}
```

第9章　AppleEvent の発行

12. ソフトウエア同士のコミュニケーションを行う AppleEvent の処理は、Mac OS 独自の処理です。あるアプリケーションから別のアプリケーションをコントロールすることに利用できます。
13. AppleScript は、Mac OS ではイベントの 1 つとして位置付けられています。その AppleScript のイベントを Java のアプリケーションから発行することが、制限はあるのものの可能です。
14. イベントの発行について、MRJ 2.1.1 での状況で説明をします。

9-1　AppleEvent について

AppleEvent の基本的なことと、Java の実行環境で、AppleEvent の扱いがどうなっているのかと言うことをまとめておきましょう。AppleEvent については概略は説明しますが、詳細については AppleEvent Manager の解説についても目を通されることを期待します。

AppleEvent の役割

ソフトウェア同士のコミュニケーションを取るための AppleEvent は、Mac OS 独自の機能です。ダブルクリックによって書類が開くようなデスクトップ機能を実現するため、Finder あるいはシステムは積極的に利用しています。模式的には、あるソフトウエアから別のソフトウエアに対するリクエストが AppleEvent です。単にリクエストを送る場合もありますが、返答を AppleEvent の形式で得られることもあります。このやりとりは、各ソフトウエアではイベントとして処理されます。マウスのクリックやキーボード作業などを「ローレベルイベント」、AppleEvent のやりとりは「ハイレベルイベント」として明確に分けてあり、イベント処理の組み込み方法も実際には違ってきています。

AppleEvent をやりとりするソフトウエアは、アプリケーションが基本になります。あるアプリケーションから別のアプリケーションをコントロールするという具合ですが、その結果、複数のアプリケーションをまたがった連係処理も可能になります。

♦AppleScript について

AppleEvent をベースにして、スクリプトプログラムの実行が可能な AppleScript という機能も利用できます。AppleScript に対応したアプリケーションの操作を、スクリプトプログラムで記述することができます。その意味では、AppleScript は広い意味での「言語」として認識されています。

この AppleScrit によるスクリプトプログラムを実行したとき、別のアプリケーションの処理は、AppleEvent に翻訳され、AppleEvent のやりとりが発生します。AppleScript プログラムのすべてのステップが AppleEvent になるわけではありませんが、別のアプリケーションのコントロールややシステム機能の利用など、特徴的な処理を行う場面で実際には AppleEvent を利用しています。

AppleEvent の Java での利用

こうした Mac OS 独自の機能である AppleEvent は、もちろん、Java の標準機能としては利用できるようにはなっていません。しかしながら、書類を Finder でダブルクリックして開いたり、あるいは書類をドラッグ&ドロップして開くような操作は、背後で AppleEvent のやりとりが行われることになります。Mac OS のアプリケーションとしてはこうした処理を組み込むことが必須になっています。そこで、MRJ では、MRJToolkit として、必ず対応しなければならない AppleEvent の処理を、簡単に組み込めるように機能を提供しています。AppleEvent ということは意識しなくても、指定したとおりにイベントリスナーをインプリメントしたクラスを定義し、そのクラスのインスタンスを登録することで、書類を開くなどの基本的なイベントに対応できるというわけです。これについては、第 3 章で詳細を説明してあります。

こうした基本機能に加えて、MRJ 2.1 より「MRJScripting」という機能が加わりました。これは、MRJ 環境で実行するアプリケーションやあるいはアプレットを自動的に AppleScript 対応にしてしまう機能が中心です。Java で作ったプログラムを、AppleScript でコントロールするというわけです。これについては次の第 10 章で具体的に説明をします。

♦AppleEvent の発行

以上のような、MRJToolkit や MRJScripting の機能は、いわば MRJ によって正式にサポートされた機能です。これに対して、MRJ 内部の公開されていない機能を使うことによって、AppleEvent の発行もできなくはありません。ただし、発行はアプレットからはできなくなっており、アプリケーションとして作る必要があります。

そして、Java で作ったアプリケーションから、Finder やあるいは別のアプリケーションに対して、AppleEvent を発行することができます。具体的には後で説明しますが、このことによってどんなことができるようになるかを説明しておきましょう。

まず、1 つの大きな要望は、別のアプリケーションを起動して、そのアプリケーションでファイルを開かせたいということがあるでしょう。これも、もちろん AppleEvent を Finder および起動したアプリケーションに対して発行することで不可能ではありませんが、むしろ Runtime クラスの exec メソッドを使った方が手軽に作成できます。MRJToolkit の機能を使ってアプリケーションのパスを得て、あとは開くファイルのパスと含めて String 型の配列を作り exec メソッドを実行します。標準ライブラリにある Runtime クラスを使っているからとは言え、その部分は Mac OS でしか機能しない

プログラムにはなってしまいます。いずれにしても、アプリケーションの起動とファイルを開くことはわざわざ AppleEvent を使うことはないでしょう。

後のサンプルで実際に作ったものとしては、「指定した URL」を開くという機能があり、これは AppleEvent を Finder に対して発行して実現しています。この機能は、AppleScript で言えば、「open location "URL"」の機能に相当します。アプレットでは showDocument メソッドで指定したアドレスをブラウザで開くことができますが、このメソッドはアプリケーションでは使えません。そこで、アプリケーションでアドレスをブラウザで開くのに、AppleScript の拡張命令を利用するというわけです。具体的には AppleEvent を定められた形式で発行するということが必要になります。

♦Java で利用できない機能

ただし、AppleScript のすべての機能が使いやすいようになっているわけではありません。まず、AppleScript のテキストを与えて、コンパイルし実行するという機能が OSADoScript などの API コールとして Toolbox には用意されてます。しかしながら、MRJ のクラスの中には現状ではそれに相当する機能を持ったクラスはありません。また、AppleScript を発行する時にテキストやエイリアスなどの基本的なデータは記述できますが、ウインドウなどのオブジェクトデータを記述するのに必要な CreateObjSpecifier の API コールも用意されていません。

これらの機能はシステムそのものというよりも、ライブラリで提供されている機能で、C 言語で開発していても、ライブラリをプロジェクトに登録しないと実際には使えない機能です。逆に言えば、Java のネイティブコール機能を使ってライブラリをロードして呼び出せば使えるのかも知れませんが、本来は必要のあるものはクラス化して提供されるはずなので、MRJ の機能が増えるのを待つのが基本だと思われます。

MRJ のクラスとして、com.apple.mrj.MRJCoercionHandler というものがあり、内容を見る限りは、変換クラスを定義できるのではないかと思われますが、MRJScripting の機能を使う上ではそこまでの機能の作り込みは必要になることはまずないかと思われます。

AppleEvent の構造

AppleEvent の基本的な概念をまとめておきましょう。詳細については、Inside Macintosh の Interapplication Communication などを参照してください。

まず、AppleEvent は、なんらかの形でデータのやりとりを行います。そのデータを

単にやりとりするのではなく、データを「ディスクリプタ」というものでパッケージ化します。ディスクリプタは、データに加えて、データの形式を示す情報（データタイプ）が付加されたものです。つまり、整数なのかテキストなのかとようなデータの種類をデータにくっつけたものがディスクリプタです。

OS 機能としては、ハンドルの先にあるデータを管理するディスクリプタの構造体を定義して、それを処理することになります。そのため、メモリ管理は飛躍的に複雑化します。ディスクリプタの構造体、データの構造体などが絡み合って来るわけです。しかしながら、Java ではそうした心配はあまりありません。「ディスクリプタ」というオブジェクトを扱うクラスが定義されており、コンストラクタを利用すれば、データを含むディスクリプタを生成できるため、その意味では非常に素直で見通しの良いプログラムが記述できます。こうした AppleEvent の構造体はオブジェクト指向的に作られているのですが、それを C で処理していることからプログラムが複雑化していたと言えるでしょう。

ディスクリプタが AppleEvent で使うデータの基本クラスとも言えるもので、それから派生したいくつかのクラスを利用します。まず、ディスクリプタのリストがあります。これは複数の基本的には同一形式のディスクリプタが複数集まったものです。

また、複数のディスクリプタが集まりながら、個々の要素をキーワードで区別できるようにしたものを「レコード」と呼びます。レコードは、ディスクリプタのリストの発展したものとして定義されています。

そして、一定の規則に従ったレコードが、AppleEvent になります。レコードは汎用的なデータの集合ですが、たとえば、送付先や属性などの定められた情報を含むレコードが AppleEvent として送付できるわけです。クラス定義の継承の関係は次のようになります。

ディスクリプタ→ディスクリプタのリスト→レコード→AppleEvent

従って、AppleEvent は特別なデータ形式にそったディスクリプタであるということにもつながります。AppleEvent を発行する場合は、こうしたクラス階層を順番に辿ることになります。

9-2　AppleEvent ディスクリプタの処理

AppleEvent でのデータを記述するためのデータ構造であるディスクリプタの作り方や処理方法を説明しましょう。C では複雑だったこうしたデータ処理も、オブジェクト指向の Java では極めて見通しよくプログラムを作成することができます。

データタイプの記述

ディスクリプタは、データとタイプがセットになったものです。データはオブジェクトとして記述するとして、タイプについての記述方法をまず知っておく必要があります。ディスクリプタのタイプは、Toolbox 本来の定義では 4 バイトのデータです。つまり、ファイルタイプやクリエイタなどと同じ形式のアルファベット 4 文字をベースにしたデータになっています。ファイルタイプなどはそれを記述するクラスが定義されていましたが、ディスクリプタタイプについては定義されておらず、Java の基本データ型である int を利用します。

ここで、ディスクリプタのデータタイプなどの 4 バイトのタイプデータは、たとえば「MACS」のような通常はテキストとして記述できる形式で定義されており、AppleEvent Registray などを参照したときには、その 4 バイトデータを 4 文字の文字列でリファレンスできるようになっています。4 文字データを int をいちいち変換するのは不便ですので、com.apple.MacOS.OSUtils クラス、つまり OSUtils クラスに定義されている以下のメソッドを使うと便利でしょう。

表 9-1　データタイプの int 型データを生成するメソッド

戻り値	メソッド	機能と使い方
int	OSUtils.makeOSType(char,char,char,char)	引数の 4 つの文字を持つタイプを作成する【static】
int	OSUtils.makeOSType(String)	引数の文字列で形成されるタイプを作成する【static】
int	OSUtils.makeOSType(byte[])	引数のバイト値を持つタイプを作成する【static】

たとえば、ディスクリプタで 32 ビットの整数を示すタイプは、文字列で書くと「long」です。実際には、

```
int integerType = OSUtils.makeOSType("long");
```

のようなプログラムで、int 型のタイプを示すデータを作成します。

また、こうしたタイプは、たとえば C や C++でプログラムする時には、Universal Header のヘッダファイルで定義されています。Java 環境では、com.apple.MacOS.ae というクラスで定義されています。このクラスはインタフェースクラスで、ディスクリプタのクラスである AEDesc がインプリメントしているため、AEDesc あるいは継承クラスでのクラス変数として使用できます。以下のようなクラス変数が利用できますが、分かりやすくするために、AEDesc のクラス変数として記述しておきます。

表 9-2　タイプの指定に利用できる定義定数

クラス変数	値	意味
AEDesc.typeWildCard	****	どのタイプにも合う【static】
AEDesc.typeType	type	イベントクラス、イベント ID【static】
AEDesc.typeBoolean	bool	boolean 型【static】
AEDesc.typeTrue	true	boolean 型の true 値【static】
AEDesc.typeFalse	fals	boolean 型の false 値【static】
AEDesc.typeChar	TEXT	テキスト【static】
AEDesc.typeInteger	long	32 ビットの整数値【static】
AEDesc.typeSMInt	shor	16 ビットの整数値【static】
AEDesc.typeSMFloat	sing	SANE 単精度浮動小数点数【static】
AEDesc.typeShortFloat	sing	SANE 単精度浮動小数点数【static】
AEDesc.typeFloat	doub	SANE 倍精度浮動小数点数【static】
AEDesc.typeLongFloat	doub	SANE 倍精度浮動小数点数【static】
AEDesc.typeFixed	fixd	固定小数点数【static】
AEDesc.typeComp	comp	SANE 複素数【static】
AEDesc.typeExtended	exte	SANE 拡張精度浮動小数点数【static】
AEDesc.typeMagnitude	magn	符号無し 32 ビット整数【static】
AEDesc.typeAEList	list	ディスクリプタのリスト【static】
AEDesc.typeAERecord	reco	レコード【static】
AEDesc.typeFSS	fss	FSSpec 型のファイル指定データ【static】
AEDesc.typeAlias	alis	エイリアス【static】
AEDesc.typeAppParameters	appa	Process Manager のパラメータ【static】
AEDesc.typeTargetID	targ	イベント送付先のターゲット ID【static】
AEDesc.typeApplSignature	sign	アプリケーションのクリエイタ【static】
AEDesc.typeProcessSerialNumber	psn	プロセスのシリアル番号【static】
AEDesc.typeAppleEvent	aevt	AppleEvent【static】
AEDesc.typeEnumerated	enum	enum データ（スクリプトでの定義定数）

クラス変数	値	意味
		【static】
AEDesc.typeKeyword	keyw	キーワード【static】
AEDesc.typeNull	null	データ無し【static】
AEDesc.typeProperty	prop	プロパティ【static】
AEDesc.typeQDRectangle	qdrt	QuickDraw の Rect 型【static】
AEDesc.typeSectionH	sect	発行と引用でのセクションハンドル【static】
AEDesc.typeSssionID	ssid	セッション ID【static】

なお、表の中の「値」はすべてアルファベット 4 文字です。3 文字のものは、最後のキャラクタがスペースです。

ディスクプリプタの作成と処理

データから、AppleScript として送付するときの基本的な形式であるディスクリプタをまずは生成する必要があります。ディスクリプタは、com.apple.MacOS.AEDesc クラスとして、MRJ で用意されています。このコンストラクタを利用すれば、文字列やファイルなどのデータをディスクリプタ化できます。

表 9-3　AEDesc クラスのコンストラクタ

コンストラクタ	引数の指定と動作
AEDesc()	単にインスタンスを生成するが、それ以上は何もしにくいので、使うことはないと思われる
AEDesc(boolean)	引数の値を持ち、タイプが AEDesc.typeBoolean のディスクリプタを作成する
AEDesc(short)	引数の値を持ち、タイプが AEDesc.typeSMInt のディスクリプタを作成する
AEDesc(int)	引数の値を持ち、タイプが AEDesc.typeInteger のディスクリプタを作成する
AEDesc(long)	引数の値を持ち、タイプが AEDesc.typeInteger のディスクリプタを作成する（結果的に 32 ビットに丸められると思われる）
AEDesc(float)	引数の値を持ち、タイプが AEDesc.typeLongFloat のディスクリプタを作成する
AEDesc(double)	引数の値を持ち、タイプが AEDesc.typeLongFloat のディスクリプタを作成する
AEDesc(File)	引数の値を持ち、タイプが AEDesc.typeFSS のディスクリプタを作成する
AEDesc(String)	引数の値を持ち、タイプが AEDesc.typeChar のディスクリプタを作成する
AEDesc(int, int)	1 番目の引数でタイプを指定し、2 番目の引数をデータと

	するディスクリプタを作成する
AEDesc(int, String)	1 番目の引数でタイプを指定し、2 番目の引数の文字列をデータとするディスクリプタを作成する
AEDesc(int, byte[])	1 番目の引数でタイプを指定し、2 番目の引数のバイト列をデータとするディスクリプタを作成する
AEDesc(int, HandleObject)	1 番目の引数でタイプを指定し、2 番目の引数をデータとするディスクリプタを作成する（HandleObject については本文を参照）

たとえば、文字列をディスクリプタ化するには次のようにします。ディスクリプタ化して AppleEvent に組み込むのですが、その時のパラメータ指定などでこうしたディスクリプタ作成を行うことになります。

```
import com.apple.MacOS.AEDesc;
        :
String theURL = "http://msyk.locus.co.jp/mjr/";
AEDesc urlDesc = new AEDesc(theURL);
```

表の一番最後にある HandleObject というのは、com.apple.memory.HandleObject として定義されたクラスで、Toolbox のハンドルデータを扱うための基本クラスとなっています。たとえば、エイリアスを管理する Alias クラスは、HandleObject を継承しています。Toolbox ではエイリアスは AliasHandle 型で管理しましたが、こうした型を Java で扱うためのものが HandleObject です。たとえば、あるファイルのパスからエイリアスを作成し、エイリアスのディスクリプタを作るには次のようになります。

```
import com.apple.MacOS.*;
        :
String filePath = "/MacHD/Working/Text.txt";
Alias fileAlias = new Alias(filePath);
AEDesc aliasDesc = new AEDesc(AEDesc.typeAlias, fileAlias);
```

エイリアスのディスクリプタを作る時、そのディスクリプタのタイプを定義定数を使うなどして指定をしてやる必要があります。

♦ディスクリプタからデータを取り出す

ディスクリプタからデータを取り出すメソッドとして、以下のようなものが利用できます。なお、データを設定するメソッドは使いやすい形では用意されていないので、作るのはコンストラクタだけで行うことになるでしょう。以下のメソッドは、たとえば AppleEvent を受け取った場合にそこからデータを取り出すような処理で利用することになるでしょう。データの取り出しでは、場合によってはデータの強制変換を行うこともあります。typeInteger のディスクリプタから toString を使って文字列で数字

を得ることもできるわけです。

表 9-4　ディスクリプタからのデータの取り出し

戻り値	メソッド	機能
int	getDescriptorType()	ディスクリプタのタイプを取得する
int	dataSize()	ディスクリプタ内のデータのサイズをバイト数で得る。ただし、"1"は 1、"漢字"は 4 になるので、UNICODE ではなく Toolbox が管理するデータのサイズになっている模様
boolean	getBooleanValue()	ディスクリプタのデータを boolean 値として得る
short	getShortValue()	ディスクリプタのデータを short 値として得る
int	getIntValue()	ディスクリプタのデータを int 値として得る
double	getDoubleValue()	ディスクリプタのデータを double 値として得る
String	toString()	ディスクリプタのデータを String オブジェクトとして得る
File	getFileValue()	ディスクリプタのデータを File オブジェクトとして得る
HandleObject	getDataHandle()	ディスクリプタのデータを HandleObject 型のオブジェクトとして得る

ディスクリプタのコピーと強制変換

ディスクリプタをコピーするメソッドや、強制変換を行うためのメソッドが以下のように定義されています。いずれも、AEDesc 型のオブジェクトに対して利用できるメソッドです。

表 9-5　ディスクリプタのコピーと強制変換

戻り値	メソッド	機能
void	Copy(AEDesc)	y.Copy(x);により、ディスクリプタ x の内容がディスクリプタ y にコピーされる。y はあらかじめコンストラクタ（引き数のないものを含む）で領域を確保しておかないといけない。
AEDesc	coerceTo(int)	ディスクリプタを引数で指定したタイプに強制変換する

ディスクリプタのリスト

ディスクリプタのリストは、AEDesc を継承した com.apple.MacOS.AEDescList クラ

スを使って作成することができます。コンストラクタは以下の通りです。

表 9-6　AEDeskList **クラスのコンストラクタ**

コンストラクタ	機能
AEDescList()	ディスクリプタのリストを生成する
AEDescList(AEDesc)	ディスクリプタのリストを生成し、引数のディスクリプタを 1 つの要素とする

AEDescList 関連の処理メソッドとしては以下のようなものがあります。AEDescList は、たとえば、ファイルを開く OpenDocuments イベントを作成するとき、開くファイルを指定することなどに使います。コンストラクタで生成をし、putElements を使ってファイルのディスクリプタを 1 つ 1 つリストに追加するような使い方をします。

表 9-7　AEDescList **の処理メソッド**

適用	戻り値	メソッド	機能
AEDesc	boolean	isList()	ディスクリプタがリストなら true、そうでないなら false を戻す
	AEDescList	getList()	ディスクリプタから、ディスクリプタのリストを取り出す
AEDeskList	String	asString()	テキスト化だが、内容まではテキストにならない
	Int	countElements()	リストの要素の個数を求める
	Int	getElementType(int)	引数で指定した番号のディスクリプタのタイプを戻す。最初の要素は 1 を指定する
	AEDesc	getElement(int)	引数で指定した番号のディスクリプタを戻す
	AEDesc	getElement(int, int)	1 番目の引数で指定した番号のディスクリプタを 2 番目の引数で指定したタイプで戻す
	Void	getElement(int, int, byte[])	1 番目の引数で指定した番号のディスクリプタを取り出す時、2 番目の引数で指定したタイプとして解釈し、3 番目の引数に指定した byte 配列に収める
	void	putElement(int, AEDesc)	2 番目の引数のディスクリプタをリストに追加する。1 番目の引数に何番目かを指定するが、0 を指定すると末尾に追加する

適用	戻り値	メソッド	機能
	void	putElement(int, int, byte[])	3 番目引数に指定した byte 配列のデータを、2 番目の引数のタイプであるディスクリプタとしてリストに追加する。1 番目の引数は何番目に追加するかを指定する

なお、Toolbox の API には、配列の内容をそれぞれディスクリプタリストの要素として入れたり取り出す機能が用意されています。MRJ ではそれに相当するメソッドは用意されていない模様ですが、その時に使う kAEDataArray などの定義定数だけはクラス変数に定義されています。

9-3 レコードとイベントの処理

AppleEvent のレコードについての処理と、実際イベントの発行について説明しましょう。AppleEvent 自体は一定の規則に従って作られたレコードですが、さまざまな設定を行うためのメソッドが用意されています。

AppleEvent のレコード

AppleEvent のレコードは、ディスクリプタのリストなのですが、単にディスクリプタが集合しているだけでなく、1 つ 1 つのディスクリプタに「キーワード」が付与されたリストです。単なるリストなら、何番目と指定して取り出すことしかできませんが、レコードの場合はキーワードを指定して、特定のディスクリプタを取り出すことなどができます。レコードの中で、キーワードとディスクリプタをセットにしたものを「パラメータ」と呼んでいます。パラメータも実体はディスクリプタです。

レコードは、AEDescList を継承した com.apple.MacOS.AERecord クラスとして定義されています。以下のようなメソッドが使えます。あまり機能がないように思えるかも知れませんが、実質的には、AppleEvent のクラスで定義されたメソッドを使うことになるので、AERecord では汎用的な処理しか用意されていないという具合です。

表 9-8　AERecord 関連のメソッド

適用	戻り値	メソッド	機能
コンストラクタ		AERecord()	レコードのインスタンスを生成する
		AERecord(AEDesc)	引数に指定したディスクリプタを要素として持つレコードのインスタンスを生成する
AEDesk	boolean	isRecord()	ディスクリプタがレコードなら true、そうでなければ false
	AERecord	getRecord()	ディスクリプタとして得られたデータを、レコードとして取得する
AERecord	AEDesc	getParam(int)	引数に指定したキーワードのパラメータを取得する

AEDesc	getParam(int, int)	1 番目の引数に指定したキーワードのパラメータを取得する。2 番目の引数で、ディスクリプタのタイプを指定する
int	getParamSize(int)	引数に指定したキーワードのパラメータのサイズを取得する
int	getParamType(int)	引数に指定したキーワードのディスクリプタタイプを取得する
boolean	hasParam(int)	引数に指定したキーワードが存在するかを調べる
void	putParam(int, AEDesc)	指定したディスクリプタを、引数のキーワードで、パラメータとして追加する

AppleEvent の送付先

AppleEvent の送付先を指定するためのクラスとして com.apple.MacOS.AETarget クラスが定義されています。このクラスも AEDesc を継承しており、基本的にはディスクリプタです。以下のコンストラクタを利用して、送付先を記述したオブジェクトを作成します。

表 9-9　AETarget クラスのコンストラクタ

コンストラクタ	利用方法
AETarget()	インスタンスの生成だが、基本的には使わない
AETarget(int)	引数に指定したクリエイタのアプリケーションに対してイベントを送付する
AETarget(AEDesc)	引数にディスクリプタで指定したターゲット ID やセッション ID のアプリケーションに対してイベントを送付する（詳細不明）
AETarget(ProcessSerialNumber)	引数に指定したプロセスシリアル番号のアプリケーションに対してイベントを送付する

AppleEvent の作成

AppleEvent 自体が com.apple.MacOS.AppleEvent クラスとして定義されており、このクラスは AERecord を継承しています。つまり、ディスクリプタでもありレコードでもあるというわけです。

表 9-10　AppleEvent クラスのコンストラクタ

コンストラクタ	利用方法
AppleEvent(int, int, AETarget)	1 つ目の引数にイベントクラス、2 つ目の引数にイベント ID、3 つ目のクラスにイベントの送付先を指定して、AppleEvent を作成する
AppleEvent(int, int, AETarget, int, int)	3 つ目までの引数は同上。4 つ目はリターン ID、5 つ目はトランザクション ID と思われる
AppleEvent(AppleEvent)	引数の AppleEvent を元にした AppleEvent を生成する

基本的には 1 つ目のコンストラクタを使うのが一般的でしょう。ここで、送付先は AETarget として作られたオブジェクトを指定します。イベントクラス、イベント ID は、AppleEvent を識別するためのタイプで、イベントごとにきめられたものです。定義定数もある程度は使えなくもないのですが、MRJ 環境ではクラス変数が用意されていないものも多くあり、イベントクラスや ID を直接指定することがむしろ多いでしょう。

表 9-11　イベントクラスや ID で使えるクラス変数

種類	クラス変数	値	意味
イベントクラス	AEDesc.kCoreEventClass	aevt	必須イベントのグループを示す
イベント ID	AEDesc.kAEOpenApplication	oapp	起動時に発生する
	AEDesc.kAEOpenDocuments	odoc	ファイルを開く要求
	AEDesc.kAEPrintDocuments	pdoc	ファイルを印刷する要求
	AEDesc.kAEQuitApplication	quit	アプリケーション終了の要求

このイベントクラスやイベント ID は、実際には AppleEvent を受け取るアプリケーションの aete リソースの記述に従います。aete リソースは受付可能な AppleEvent の形式を記述できるリソースということになります。C 言語などで Universal Header を使ってプログラムする上では、多くのイベントが定義定数として利用できます。一方、MRJ ではタイプを示す 4 文字のキーワードを、OSUtils.makeOSType メソッドを使って指定することが多いでしょう。また、この 4 文字のキーワードが分かっていればいいのですが、公開されていないで分からない場合には、aete リソースを編集できるリソースエディタで調べるか、MPW の DeRez のようなツールを使って定義内容を解析することになります。具体的にはサンプルプログラムで方法を説明します。

AppleEvent クラスでは以下のようなメソッドを利用して、実際にイベントを送付な

どを行います。実際に送付する前に、パラメータや属性の設定を行うのが一般的です。getKeyDesc、putKeyDesc については使用しない方がいいのではないかと思われます。

表 9-12　AppleEvent **の送付と基本情報**

戻り値	メソッド	機能
AppleEvent	send()	AppleEvent を実際に送付し、応答が戻ってくるまで待つ。応答は戻り値の AppleEvent として得られる
void	sendNoReply()	AppleEvent を実際に送付するが、応答は待たずに次の処理に移る
int	getEventClass()	イベントクラスを取得する
int	getEventID()	イベント ID を取得する
AETarget	getSource()	送られてきたイベントの場合、イベントの送付元を得る
AETarget	getTarget()	イベントの送り先を得る
AEDesc	getKeyDesc()	キーワード付きのディスクリプタを得る（詳細不明）
void	putKeyDesc(AEDesc)	引数のディスクリプタを設定する（詳細不明）
int	getReturnID()	リターン ID を得る
int	getTransactionID()	トランザクション ID を得る
boolean	isSuspended()	（詳細不明）
void	suspendEvent()	（詳細不明）
void	resupeEvent()	（詳細不明）
String	toString()	文字列化するが、あまり情報は多くない

AppleEvent **の属性とパラメータ**

AppleEvent には、属性として、AppleEvent の状態などが設定されています。属性についてもディスクリプタで取り出します。なお、Toolbox の API コールでは、属性の設定もできますが、MRJ 環境では設定するメソッドは用意されていないようです。

また、AppleScript にはパラメータを含めることができます。AppleScript では、たとえば「open theFile with theApp」のような命令を Finder に送って、指定したファイルを指定したアプリケーションで開くことができます。このとき、open に対応する指定が前に説明したイベントクラスとイベント ID です。そして、theFile や theApp の指定がパラメータに相当します。

属性やパラメータの取り出し、あるいは設定のメソッドは次のようなものを利用できます。

表 9-13　属性とパラメータの処理

戻り値	メソッド	機能
AEDesc	getAttribute(int)	引数で指定した種類の属性を取り出す
void	getAttribute(int, AEDesc)	同上。取り出した属性データは、2 番目の引数に指定したディスクリプタに設定される
void	getAttribute(int, int, AEDesc)	同上。2 番目の引数には取り出すディスクリプタのタイプを指定する。取り出した属性データは、3 番目の引数に指定したディスクリプタに設定される
AEDesc	getParameter(int)	引数で指定したキーワードのパラメータに含まれるディスクリプタを取得する
void	getParameter(int, AEDesc)	同上。取り出したディスクリプタは、2 番目の引数に指定したインスタンスに設定される
AEDesc	getParameter(int, int)	1 番目の引数で指定したキーワードのパラメータに含まれるディスクリプタを取得する。2 番目の引数には取り出すディスクリプタのタイプを指定する
void	getParameter(int, int, AEDesc)	同上。取り出したディスクリプタは、3 番目の引数に指定したインスタンスに設定される
void	putParameter(int, int, byte[])	AppleEvent にパラメータを追加する。1 つ目の引数はキーワード、2 つ目の引数はデータである 3 つ目の引数の種類をディスクリプタタイプで指定する
void	putParameter(int, AEDesc)	AppleEvent にパラメータを追加する。1 つ目の引数がキーワードで、2 つ目の引数が実際のデータを示すディスクリプタ

属性について、属性の種類を指定するクラス変数が定義されていますが、これは省略します。これらで得られるような属性は、たとえばイベントクラスなどで、必要ならそのためのメソッドを利用して取得する方が手軽だからです。クラス変数は、Toolbox の AEGetAttributePtr や AEGetAttributeDesc で使う定義定数と同じ名称のものが、ae クラスで定義されています。

パラメータについては、前の例では、theFile のように命令の後に直接記述するものがあり、これを「ダイレクトパラメータ」と呼んでいます。ダイレクトパラメータを指定するには、キーワードとして、クラス変数の AEDesc.keyDirectObject（値は '----'）を指定して putPrameter メソッドでディスクリプタを AppleEvent に追加します。たとえば、ディスクリプタには File オブジェクトを引数にとったコンストラクタの戻り値などを指定するという具合です。

ダイレクトパラメータ以外のパラメータは、AppleScript 上でも前の例の「with」の

ようなキーワードが付加されています。こうしたパラメータは、そのイベントで規定されたキーワードのタイプを指定します。これもイベント ID などと同様、イベントを受け付けるアプリケーションの aete リソースで定義されています。

ダイレクトパラメータ以外のキーワードでは、たとえば戻ってきた AppleEvent からエラーを取得するための AEDesc.keyErrorNumber や AEDesc.keyErrorString などいくつかはクラス変数が定義されていますが、基本的には OSUtils.makeOSType メソッドを使って文字列で記述することが多くなるでしょう。

具体的な AppleEvent の送付方法はサンプルプログラムのところで説明します。

9-4 サンプルプログラム

サンプルとして、URL の文字列を与えて、それをブラウザなどで開くこと、そして指定したファイルを指定したアプリケーションで開くことを行ってみます。アプリケーションで URL を開く機能は Java のライブラリでは用意されていないので、前者のようなニーズは多いかもしれません。後者のことは Runtime.exec で同等なことができますが、イベント送付のサンプルとして紹介します。

ソースファイルの OpenURL.java

AppleEvent を送付する機能を、static なメソッドに組み込みました。以下のソースでは、単にアプリケーションを起動して、それらの AppleEvent 発行メソッドを呼び出しているだけです。

doOpenURL メソッドは、引数のアドレスを開きます。http ならブラウザ、ftp なら ftp クライアントが開くようにも可能ですが、プロトコルとそれに対するアプリケーションは、システム側の設定に依存します。

doOpenFileBy は、1 つ目の引数に文字列配列で指定した複数のファイルを、2 つ目の引数で指定したクリエイタのアプリケーションで開くと言うものです。ソースファイルのコメント中にも記載しましたが、この機能はわざわざ AppleEvent を使わなくても、Runtime の exec メソッドを使うことで実現できますが、ある程度複雑なイベント処理をサンプルとして示すために作成をしました。

ソースファイルは 1 つだけで、アプリケーションとして作成します。実行を開始するクラスはもちろん、OpenURL を指定します。また、機能拡張フォルダの MRJLibraries の MRJClasses フォルダにある、MRJClasses.zip もプロジェクトに加えておきます。MRJToolkitStubs.zip では AppleEvent 関連のクラス定義がなされていないようで、これではなく MRJClasses.zip そのものをプロジェクトに追加しておく必要があります。また、Java MacOS Post Linker を使用して、Mac OS のアプリケーションとして別途 Finder から起動する方がスムーズに動作するようです。

```
import com.apple.MacOS.*;
import com.apple.mrj.*;
import java.io.*;
```

```
public class OpenURL      {

    public static void main()       {
        doOpenURL("http://www.apple.co.jp");   //引数に指定した URL を開く

        String[] openFiles = {
            "/partner/書類/MJR/MJR Samples/ch09_SendAE/OpenURL.java",
            "/partner/書類/MJR/MJR Samples/ch09_SendAE/TrivialApplication.java"};
                //開きたいファイルのフルパスを要素に持つ配列
        doOpenFileBy(openFiles, "ttxt");    //各ファイルをクリエイタが ttxt の SimpleText で開く
    }

/*  引数に文字列で指定した URL を開く。
    AppleScript で言えば、「open location "http:/..."」に相当する。
    これは、OSAX の「標準機能追加」で実現されている拡張命令。
*/
    public static void doOpenURL(String urlStr)    {
        AEDesc directParam = new AEDesc(urlStr);
                //開くアドレスの文字列をディスクリプタ化する
        AEDesc errRepParam = new AEDesc(false);
                //パラメータの error reporting に指定する論理値をディスクリプタ化する
        AEDesc finderAddress = new AEDesc(
                        //イベントの送付先は Finder にする。ただし open location は OSAX
                        //の機能なので、このイベントは何に送ってもかまわない
                AEDesc.typeApplSignature,
                        //      送付先をクリエイタで指定することを示すタイプを設定
                OSUtils.makeOSType("MACS"));        //      Finder のクリエイタ
        AETarget finderTarget = new AETarget(finderAddress);
                        //イベントの送付先を示すオブジェクトを生成
        AppleEvent sendingEvent = new AppleEvent( //Apple イベントを作成する
                OSUtils.makeOSType("GURL"),         //      イベントクラスを指定
                OSUtils.makeOSType("GURL"),         //      イベント ID を指定
                finderTarget);                      //      送付先を指定
        sendingEvent.putParameter(                  //ダイレクトパラメータをイベントに設定
                AEDesc.keyDirectObject,             //      ダイレクトパラメータを示す値を指定
                directParam);                       //      パラメータに設定する値
        sendingEvent.putParameter(              //error reporting パラメータをイベントに設定
                OSUtils.makeOSType("errr"),     //      ダイレクトパラメータを示す値を指定
                errRepParam);                   //      パラメータに設定する値
        sendingEvent.sendNoReply();             //イベントを送付する。応答はなし
    }


/*  注意：
        以下の doOpenFileBy メソッドの処理は、Runtime クラスの exec メソッドで
        まったく同様なことができます。ここでは、AppleScript の処理のサンプルとして
        このプログラムを紹介しているので、御承知おきください。

        たとえば、同等なプログラムとしては、以下のようなものになります。

        String[] openFiles2 = {
            "/partner/アプリケーション/SimpleText",
            "/partner/書類/MJR/MJR Samples/ch09_SendAE/OpenURL.java",
```

```
            "/partner/書類/MJR/MJR Samples/ch09_SendAE/TrivialApplication.java"};
        try    {
            Runtime.getRuntime().exec(openFiles2);
        }    catch(Exception e)
            {    System.out.println(e.getMessage());    }
*/

/*  引数 openFiles で String の配列を指定する。配列の各要素は、ファイルへのフルパス
    それらのファイルを、引数 opener で指定したクリエイタを持つファイルで開く
*/
    public static void doOpenFileBy(String[] openFiles, String opener)    {

        //引数 opener に指定したクリエイタのアプリケーションを起動する
        File openerFile = null;
        try    {
            openerFile = MRJFileUtils.findApplication(
                        new MRJOSType(opener));
                //クリエイタが opener のアプリケーションファイルを見つける
        }    catch(Exception e)
            {    System.out.println(e.getMessage());
                return;    }    //ない場合などは処理を中止

        System.out.println("Launching: "+openerFile.toString());
                                    //起動するアプリケーションを標準出力に表示
        AEDesc finderAddress = new AEDesc(        //イベントの送付先は Finder にする
                AEDesc.typeApplSignature,
                        //    送付先をクリエイタで指定することを示すタイプを設定
                OSUtils.makeOSType("MACS"));        //    Finder のクリエイタ
        AETarget finderTarget = new AETarget(finderAddress);
                                        //イベントの送付先を示すオブジェクトを生成
        AppleEvent sendingEvent = new AppleEvent( //AppleEvent を生成する。
                        //    指定したファイルを Finder で開く（Open Selection）
                OSUtils.makeOSType("FNDR"),        //    イベントクラスを指定する
                OSUtils.makeOSType("sope"),        //    イベント ID を指定する
                finderTarget);        //    送付先は Finder
        Alias folderAlias = new Alias(new File(openerFile.getParent()));
                //開くアプリケーションが存在するフォルダのエイリアスを作成
        AEDesc directObj = new AEDesc(AEDesc.typeAlias, folderAlias);
                //エイリアスをディスクリプタ化する
        sendingEvent.putParameter(AEDesc.keyDirectObject,directObj);
                //それをダイレクトパラメータに指定する

        Alias targetAlias = new Alias(openerFile);
                //開くアプリケーションファイルのエイリアスを作成
        AEDesc selObj = new AEDesc(AEDesc.typeAlias, targetAlias);
                //エイリアスをディスクリプタ化する
        AEDescList targetList = new AEDescList();    //ディスクリプタのリストを生成し
        targetList.putElement(0, selObj);                //ファイルをリストに追加する
        sendingEvent.putParameter(OSUtils.makeOSType("fsel"),targetList);
                //リストをパラメータに指定する

        sendingEvent.send();  //AppleEvent として送付する

        //起動したアプリケーションに対して、OpenDocuments イベントを送る
```

```
            AEDesc appAddress = new AEDesc(
                                        //イベントの送付先は開いたアプリケーションにする
                        AEDesc.typeApplSignature,
                                    //      送付先をクリエイタで指定することを示すタイプを設定
                        OSUtils.makeOSType(opener));
                                    //      開いたアプリケーションのクリエイタ
            AETarget appTarget = new AETarget(appAddress);
                                    //イベントの送付先を示すオブジェクトを生成
            sendingEvent = new AppleEvent(          //AppleEvent を生成する
                        AEDesc.kCoreEventClass,         //      イベントクラス。クラス変数を使える
                        AEDesc.kAEOpenDocuments,    //      イベント ID
                        appTarget);
            targetList = new AEDescList();      //ディスクリプタのリストを作成
            for (int i=0 ; i<openFiles.length ; i++)    {       //引数 openFiles の各要素について
                  selObj = new AEDesc( //そのファイルのエイリアスを作成しディスクリプタ化する
                              AEDesc.typeAlias,
                              new Alias(new File(openFiles[i])));
                  targetList.putElement(0, selObj);   //ディスクリプタをリストに追加
            }
            targetList.putElement(0, new AEDesc(123456));

            sendingEvent.putParameter(AEDesc.keyDirectObject, targetList);
                                    //作成したディスクリプタのリストをダイレクトパラメータに指定
            AppleEvent returnEvent = sendingEvent.send();           //イベントを送付する

            System.out.println(returnEvent.toString());
                                    //戻ってきたイベントを標準出力に書き出す

//    System.out.println((returnEvent.getParameter(AppleEvent.keyErrorNumber)).toString());

        }
}
```

doOpenURL の設計と動作

指定したアドレスを開くというのは、AppleScript では、open location コマンドで実現されています。OSAX として提供されいてるコマンドで、Mac OS 8.5 では、「スクリプティング機能追加」フォルダの「標準機能追加」というファイルで、コマンドの機能が提供されています。コマンドの書式などは、用語説明で確認できます。

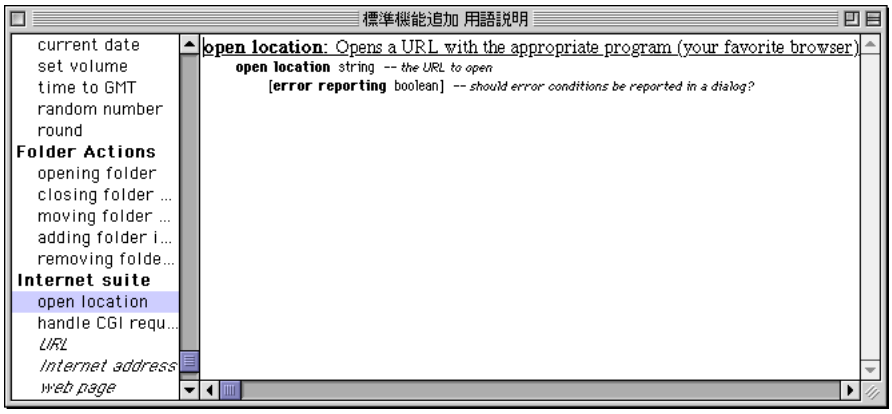

図 9-1　　open location **の用語説明**

ただし、スクリプトの用語説明だけでは、これに対応する AppleEvent を発行する情報には足りません。そこで、「標準機能追加」ファイルの aete リソースを逆コンパイルして調べることにします。たとえば、MPW で DeRez というコマンドを使えば、リソースの定義の形式で得られます。このとき、AEUserTermTypes.r などのいくつかの AppleEvent 関係のインクルードファイルをコマンド実行時に指定する必要があります。得られた結果から、コマンド名「open location」を検索して、その定義部分を探します。以下のようになっています。得られた結果にポイントを追記しました。(aete リソースを逆コンパイルすることはリバースエンジニアリングには相当しないと理解します。そのことがリバースエンジニアリングに当たり制限されるのなら、用語説明を見ることも同様になると考えられるからです。)

```
/* [9] */
"Internet suite",
"Standard terms for Internet scripting",
'gurl',
1,
1,
{       /* array Events: 2 elements */
        /* [1] */
        "open location",                                ←コマンド名
        "Opens a URL with the appropriate program"
        " (your favorite browser)",
        'GURL',                                 ←イベントクラス
        'GURL',                                 ←イベント ID
        noReply,                                ←コマンドの戻り値がない
        "",
        replyRequired,
        singleItem,
        notEnumerated,
        notTightBindingFunction,
        reserved,   reserved,   reserved,   reserved,   reserved,   reserved,
```

```
                reserved,  reserved,  verbEvent, reserved,  reserved,  reserved,
                'TEXT',                          ←ダイレクトパラメータのデータ形式
                "the URL to open",
                directParamOptional,             ←省略可能であることを示す
                singleItem,
                notEnumerated,
                doesntChangeState,
                reserved,  reserved,  reserved,  reserved,  reserved,  reserved,
                reserved,  reserved,  reserved,  reserved,  reserved,  reserved,
                {       /* array OtherParams: 1 elements */
                        /* [1] */
                        "error reporting",       ←キーワード付きパラメータのスクリプトでのキーワ
ード
                        'errr', ←キーワード付きパラメータの AppleEvent でのキーワード
                        'bool',      ←キーワード付きパラメータのデータ形式
                        "should error conditions be reported in a"
                        " dialog?",
                        optional,                ←省略可能であることを示す
                        singleItem,
                        notEnumerated,
                        reserved,  reserved,  reserved,  reserved,  reserved,  reserved,
                        reserved,  reserved,  reserved,
                        prepositionParam,
                        notFeminine,
                        notMasculine,
                        singular
                },
```

以上のように、イベントクラス、イベント ID がいずれも「GURL」であることが分かります。AppleEvent クラスのオブジェクトをインスタンス化するときに、イベントクラスと ID を指定しますが、この値を OSUtils.makeOSType("GURL") のようにして指定することになります。

開く URL は、文字列型のデータとして、ダイレクトパラメータで指定してやります。プログラムでは、まず、URL の文字列を引数にした AEDesc のコンストラクタを呼び出してインスタンスを生成します。そして、生成した AppleEvent のオブジェクトに対して putParameter で文字列のディスクリプタを、ダイレクトパラメータのキーワードを付けて AppleEvent に付加します。

同様に、error reporting のパラメータは、'bool' つまり boolean 型のディスクリプタを生成し、それを aete から読み取った 'errr' というキーワードで、putParameter を使って AppleEvent に付加します。

このイベントは、OSAX で提供されているので、どのアプリケーションに対して送ってもよいのですが、ここでは Finder に対してイベントを送付します。イベントの送付先として、クリエイタを利用しますが、Finder のクリエイタは MACS です。まず、クリエイタのディスクリプタを作りますが、ディスクリプタタイプは

AEDesc.typeApplSignature になります。そして、その作ったディスクリプタを元に、AETarget クラスのインスタンスを生成します。それを AppleEvent のインスタンスを生成する時に指定します。

doOpenFileBy の動作と設計

doOpenFileBy メソッドでは、2 回の AppleEvent の送付を行っています。まず、指定したクリエイタのアプリケーションを開くイベントを Finder に対して行います。そして、開いたアプリケーションに対して、書類を開くイベントを送付しています。

まず、指定したクリエイタのアプリケーションを開くのですが、そのアプリケーションファイルのパスを知る必要があります。この機能は、MRJToolkit に含まれており、MRJFileUtils.findApplication というメソッドを利用して得ることができます。このアプリケーションファイルを Finder に開かせることで、まずはアプリケーションを起動することにします。

そのアプリケーションファイルを開くという作業では、Finder がサポートする Open Selection というイベントを利用します。これは、AppleEvent Registry というドキュメントを見れば、こうしたイベントが書かれているのですが、AppleScript の命令に対応したイベントが多いものの、必ずしも 1 対 1 でもないようです。Open Selection のイベントは、AppleScirpt では使われていないようです。AppleEvent Registry から、Open Selection イベントは次のような仕様になっています。

表 9-14　Open Selection イベントの仕様

イベント名	Open Selection	（戻り値）	なし
イベントクラス	(kAEFinderEvents)	= 'FNDR'	
イベント ID	(kAEOpenSelection)	= 'sope'	
パラメータ	キーワード	ディスクリプタのタイプ	内容
	AEDesc.keyDirectObject = '----'	AEDesc.typeAlias	開く項目が存在するフォルダのエイリアス（ダイレクトパラメータ）
	(keySelection) = 'fsel'	AEDesc.typeAEList	開く項目のリスト。リストの各項目はエイリアス

表中の () は、C 等のプログラミング環境で利用される定義定数名で、MRJ 環境ではクラス変数が定義されていないものです。

開くアプリケーションのファイルが得られたら、そのファイルとそのファイルが存在するフォルダのエイリアスを作成し、それらをさらにディスクリプタ化します。一方、AppleEvent は表に示したイベントクラスと ID を指定して作成します。作成した AppleEvent に、putParameter を使ってパラメータを指定します。フォルダのエイリアスをダイレクトパラメータに、ファイルのエイリアスを 'fsel' というキーワードを指定して追加します。

次に、開いたアプリケーションに対して、Open Documents イベントを発行して、実際に文書を開かせます。起動したアプリケーションによってサポートするイベントは異なりますが、このイベントは必須イベントなので、ファイルを開くと言う仕様についてはどのアプリケーションも基本的には同じはずです。

表 9-15　Open Documents **イベントの仕様**

イベント名	Open Documents		
イベントクラス	AEDesc.kCoreEventClass	= 'aevt'	
イベント ID	AEDesc.kAEOpenDocuments	= 'odoc'	
パラメータ	キーワード	ディスクリプタのタイプ	
	AEDesc.keyDirectObject = '----'	AEDesc.typeAEList	開く項目のリスト。リストの各要素は開く項目のエイリアスのディスクリプタ
戻り値	AEDesc.keyErrorNumber = 'errn'	AEDesc.typeLongInteger	エラー番号
	AEDesc.keyErrorString = 'errs'	(typeIntlText) = 'itext'	エラーを説明する文字列

Open Documents イベントは、戻り値があります。つまり、イベントを送ったアプリケーションが返答として AppleEvent を戻します。send メソッドの戻り値から、戻された AppleEvent を参照できます。戻されるイベントはエラーがある時の番号とメッセージですので、エラーがなければ戻されるイベントにはパラメータは存在しません。

開くファイルの指定では、1 つ 1 つのファイルについて、まず File オブジェクトを生成し、それからエイリアスのオブジェクトを生成し、それを AEDescList に逐一追加するという処理を行っています。具体的にはプログラムのコメントを参照してください。

（第 9 章以上）

第10章 AppleScript 対応

● Java で作成したアプリケーションは、AWT のコンポーネントを利用すれば、そのままでスクリプト対応になります。

● また、アプレットについても、Apple Applet Runner で実行する上ではスクリプト制御が可能になります。

● これらは、MRJ 2.1 で搭載された MRJ Scripting の機能を利用することで実現します。MRJ Scripting を考慮した、Java アプリケーションでのスクリプト利用について解説を行います。

10-1 AppleScript と Java

Java のアプリケーションやアプレットと AppleScript の関係についての概略をまとめておきましょう。アプレットも Applet Runner を利用するとスクリプト可能になりますが、むしろ実用的なのはアプリケーションでしょう。また、JDirect2 の機能を使って Toolbox の API を利用することでスクリプト対応にするという方法もとれますが、これについては情報源だけを示しておきます。

MRJ Scripting の概要

MRJ 2.1 から搭載された、MRJ Scripting の機能は、Mac OS 環境で機能する Java のソフトウエアを、AppleScript などのスクリプトによって処理できるようにする機能です。正しくは、OSA（Open Scripting Architecture）にのっとったスクリプティングシステムであれば何でも使えるのですが、事実上、AppleScript だけしかないと言えるので、MRJ Scripting は、Java のソフトウエアを AppleScript で制御するものと限定してもかまわないでしょう。

Java のソフトウエアを AppleScript で制御するということは、つまりは Java のソフトウエアにあるオブジェクトが、AppleScript でのオブジェクトとして扱えることになります。つまり、Java の世界が、AppleScript の世界として見えるような機能を提供しているということになります。具体的には次節で説明しますが、Java のオブジェクトが AppleAScript でのオブジェクトになり、メソッドはプロパティやコマンドになります。こうした AppleScript 利用のために必要なことは、用語集のもとになる aete リソースをアプリケーションに含めることだけで、スクリプト処理に対応するようなプログラムの追加は基本的に必要ありません。つまり、Java の世界が AppleScript に開放されるようなものと考えれば良いでしょう。

機能的には、AppleEvent が Java でのメソッド呼び出しにシステム側で翻訳されると考えることもできます。AppleScript は内部的には AppleEvent として処理されます。AppleEvent のデータを、Java のデータに変換する等の処理を、MRJ Scripting がこなしているというわけです。

なお、MRJ Scripting については、MRJ SDK の MRJ Scripting フォルダにある About

MRJ Scripting というドキュメントか、Technote 1162「Introduction to MRJ Scripting with AppleScript fo Java」に詳しく説明されています。両者の内容はおおむね同じになっています。

アプレットをスクリプト処理

MRJ Scripting の 1 つの大きな機能として、アプレットを AppleScript でコントロールできるようになるという点があります。ただし、どんなブラウザで動いていてもいいわけでなく、Apple Applet Runner で起動したアプレットのみとなっています。

アプレットについても、Java の世界が AppleScript でそのまま制御できます。Apple Applet Runner を使う必要はあるものの、Sun がサンプルとして提供している各種のアプレットを AppleScript でコントロールする方法が、前出のドキュメントに解説されています。もちろん、Sun のサンプルですしアプレットなので特に AppleScript 対応したプログラムが含まれているというわけではありません。それでも、AppleScript でコントロールができるというわけです。aete など Mac OS で必要とするリソースは、Apple Applet Runner が所持しているので、JAR あるいは CLASS ファイルのアプレットがそのままスクリプタブルになるというわけです。

ただ、この機能は面白いとは思うのですが、Apple Applet Runner で実行する必要があり、応用面では限られます。もちろん、MRJ Scripting のデモンストレーションなどには適しているかもしれませんが、一般的な用途としては扱いにくいでしょう。MJR ではこうしたこともできると言うことだけにとどめておきますので、詳細は前出のドキュメントないしは、MRJ SDK の MRJ Scripting フォルダにある About Scripting Applet Runner というドキュメントを参照してください。

AppleEvent 受信可能なアプリケーション

C や C++などでアプリケーションを作った場合、AppleScript 処理を可能にするためには、AppleEvent を受信可能なようにプログラムを作り、リソースなどを整えることになります。もちろん、PowerPlant のようなライブラリを使えばそこで用意されている機能を利用できるものの、基本的にはスクリプト処理のための追加プログラムが必要になります。その概要は次の通りです。

まず、AppleEvent を受信する関数を定義します。そこでは、パラメータなどの処理や実際の動作などを定義することになります。そして、その関数を、イベントハンドラーのテーブルに登録します。つまり、ある特定のイベントがやってきたとき、ハン

ドラーを参照して適切な関数を呼び出すという仕組みになっています。そうした登録作業を行っておくわけです。

こうしたやりとりを、Java のアプリケーションでも作成は可能です。Toolbox の API を呼び出す JDirect2 の機能を使って、イベントハンドラにメソッドを登録するというわけです。MRJ Scripiting の機能は、Java のアプレットやアプリケーションに展開された AWT のオブジェクトをベースに機能するため、単にコマンドを送るというような処理は制約が多く、事実上できません。一方、JDirect2 を使って Toolbox の機能でスクリプト対応すると、単なるコマンドなど柔軟な処理に対処できます。ただし、そのために追加のプログラムが必要にはなります。MJR では MRJ Scripting を中心に説明します。JDirect2 を使った方法については、Apple から配付されているサンプルプログラムの「AppleEvent Send and Receive」にサンプルがあるので、それを参照してください。

10-2 MRJ Scripting によるスクリプト処理

Java で作ったアプリケーションが、AppleScript の世界、具体的にはスクリプト編集プログラムからどのように見えるのかを説明しましょう。ここで利用したプログラムは 10-4 でソースを示す通り、通常の Java アプリケーションです。それがそのままスクリプト処理できるところを解説します。

aete リソースを含むプロジェクト

スクリプト処理が可能な Java アプリケーションを作る場合、基本的な aete リソースをアプリケーションに追加しておくのが便利です。最終目標では必要な aete リソースだけを用意することにしたいところですが、最初の段階では、Java のコンポーネントに対する aete リソースをアプリケーションに付加しておくのが便利です。

そのための aete リソースが、MRJ SDK の MRJ Scripting フォルダにある MRJ Scripting aete というファイルです。このファイルをプロジェクトに追加しておきます。また、追加したら、プロジェクトインスペクタを表示して、「生成ファイルにマージ」のチェックボックスを入れておきます。プロジェクトインスペクタは、プロジェクトウインドウのボタンをクリックするか、あるいは、「ウインドウ」メニューの「プロジェクトインスペクタ」を選択して表示します。

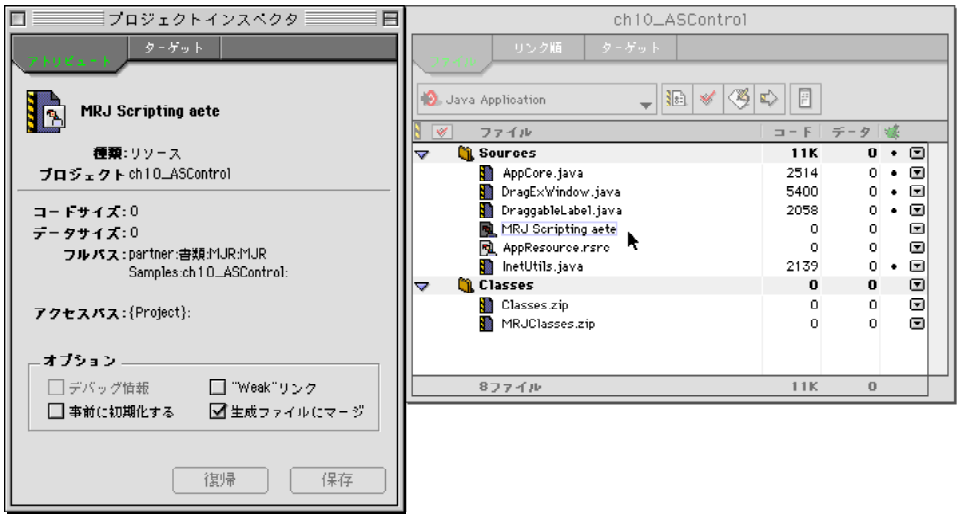

図 10.1　MRJ Scripting aete **を含めたプロジェクト**

この aete リソースを含むようにしておけば、あとは通常通りのプロジェクトです。Java のソースファイルに加えて、Classes.zip、MRJClasses.zip が登録されている必要があります。また、ターゲットの設定では、Java MacOS Linker をポストリンカに選択しておき、Mac OS でのアプリケーションを生成させます。Metrowerks の実行環境で動かしてもスクリプト処理はできません。独立したアプリケーションでなければ意味はないので、動作チェックなども生成したアプリケーションをダブルクリックして起動して行うことになります。

アプリケーションを AppleScript から見る

こうして作成したアプリケーションの用語説明をスクリプト編集プログラムで参照してみます。ここでは、DragEx という名前のアプリケーションを作成しましたが、そのアプリケーションをスクリプト編集プログラムにドラッグ&ドロップすれば、用語説明を参照することができます。いきなり、たくさんの用語がでてきますが、これは、MRJ Scripting aete というリソースファイルのリソースによって得られる用語です。

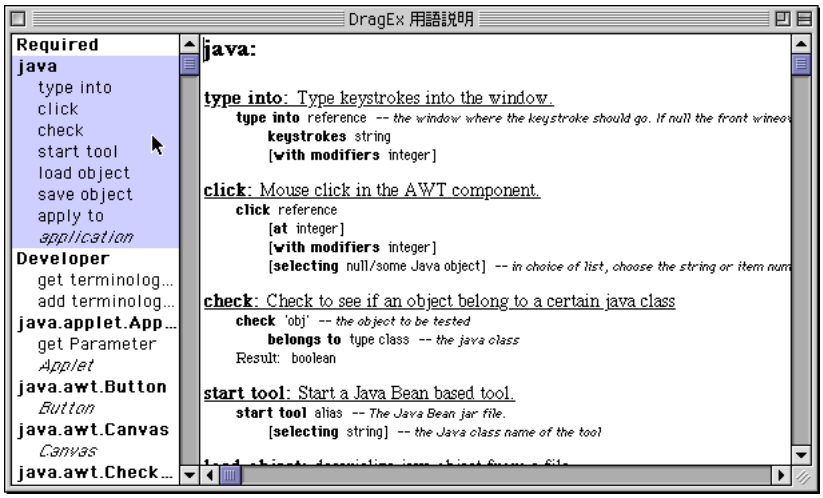

図 10.2　**作成したアプリケーションの用語説明を開いた**

用語説明をざっと見てみましょう。まず、Java というスイートには、キータイプやクリック作業を発生させるコマンドなどが並んでいます。また、Developper というスイートがあります。これらは後で説明をしましょう。

それ以降は、java.awt.Button など、Java の標準ライブラリにあるクラス名が並んでいます。おおまかに言えば、クラスが 1 つのスイートとなり、その中でクラスと同等のオブジェクトが定義され、さらにスイート内ではそのクラスのメソッドがコマンドとして定義されているというわけです。よく利用するのは、やはり Component クラスだと思いますが、このクラスの AppleScript でのオブジェクトを用語説明で見ると、いくつかのプロパティが定義されていることが分かります。

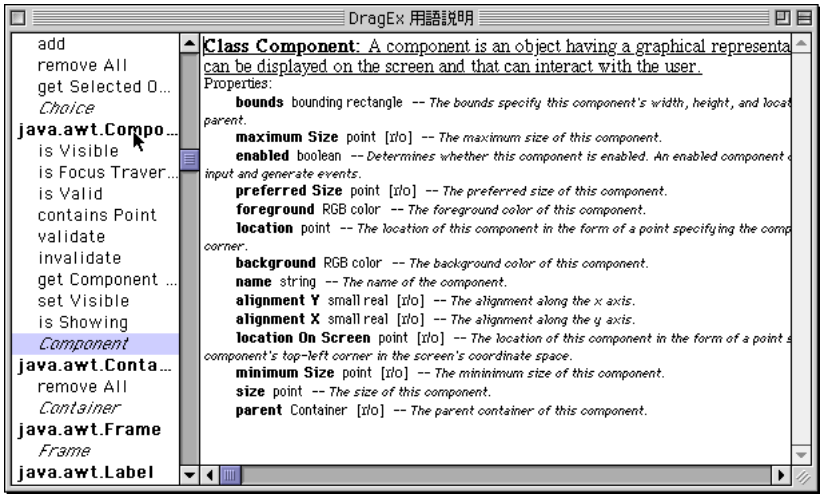

図 10.3　Component **オブジェクトのプロパティ**

スクリプト可能なオブジェクト

ここの説明に使っているサンプルプログラムは、ウインドウを表示し、ドラッグして移動可能なラベルを配置するというものです。ラベルは「メニュー」メニューから「ラベル作成」を選択すると、ウインドウ内に作成されます。ウインドウ内には、テキストフィールドも配置していますが、これは移動できません。いずれも、AWT のコンポーネントを使ったシンプルなアプリケーションです。

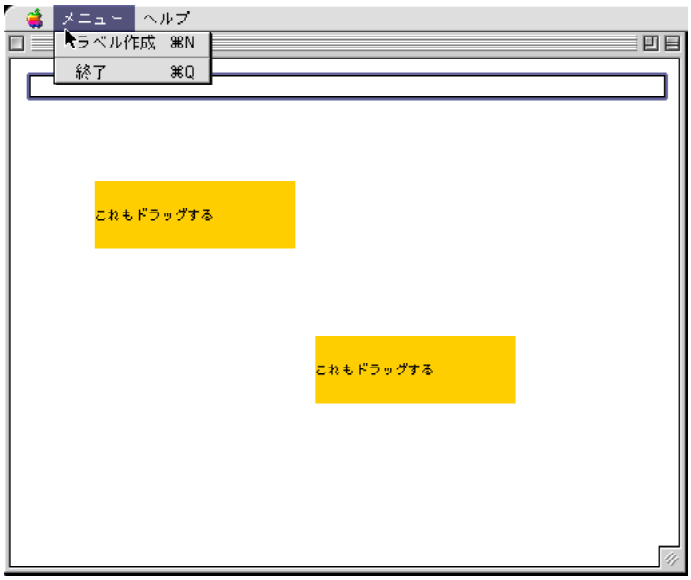

図 10.4　ラベルをいくつか配置したアプリケーション

10-4 で紹介するサンプルプログラムを実際に確認しながら結果を見てほしいのですが、このサンプルでは、ウインドウの表示のために、DragExWindow という Frame クラスを継承したクラスを定義しています。また、ドラッグ可能なラベルは、Label クラスを継承した DraggableLabel というクラスを定義しています。以上のように、それぞれ定義したクラスのインスタンスを生成した段階で、あらためて用語説明を見てみます。スクリプト編集プログラムで用語説明が開いているのなら、いったん閉じて再度開いてみます。すると、次のように、アプリケーションのプログラムで定義したクラスについての用語が加わっています。

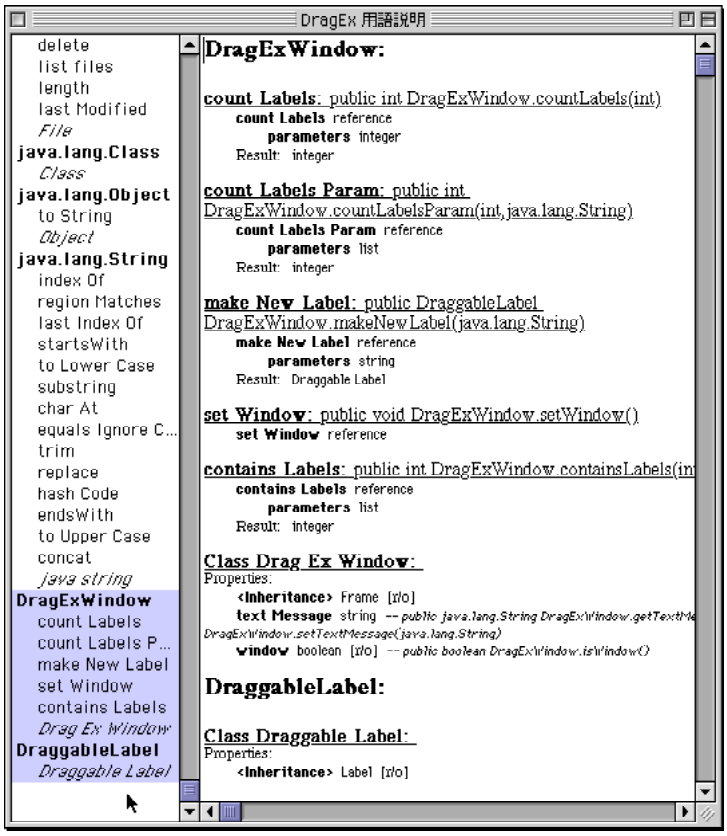

図 10.5　**生成したインスタンスのクラスが追加された**

これら、DragExWindow や DraggabelLabel に対する aete リソースは用意していません。それなのに、用語説明で出てくるのは、MRJ が内部でダイナミックに生成されたインスタンスのクラスに対する用語データを生成するからです。ここでは、Java の標準ライブラリにある Reflection の機能を使って、クラスに存在するメソッドなどを調べて、そこから aete リソースのデータを自動的に構築するわけです。

認識される AWT のオブジェクト

このアプリケーションの場合は、まず、ウインドウがあってそれは DragExWindow クラスです。このクラスは用語説明を見れば、AppleScript の世界では「Drag Ex Window」というように、大文字の前にスペースが入ったクラス名になります。そして、実際に存在するオブジェクトを参照するには、クラス名の後に 1 以上の整数を付ける方法を用います。つまり、「Drag Ex Window 1」というのが、開いているウインドウ（通常はアクティブウィンドウ）を参照する記述となるわけです。¥

また、オブジェクトは名前でも参照できます。たとえば、「Drag Ex Window "frame0"」のような形式です。この名前は自動的に付けられており、name プロパティで知るが可能です。明示的に名前を付けたいのであれば、Java のアプリケーションの側でそのオブジェクトに対して setName メソッドで名前を与えておきます。すると、AppleScript 側ではその名前で参照することができます。

たとえば、次のようなプログラムで、ウインドウのタイトルを「AppleScript and Java」に変更します。用語説明の Frame クラスを参照すると、title というプロパティがあることが確認できます。Drag Ex Window は Frame を継承していることも、用語説明で出てきています。

```
tell application "DragEx"
    set title of Drag Ex Window 1 to "AppleScript and Java"
end tell
```

ウインドウ内にあるコンポーネントは、Component クラスを使って参照することができます。以下のプログラムは、1 つ目の Component の位置を (100, 100) に変更します。Component クラスでは location というプロパティがあることが用語説明で参照できます。これはオブジェクトの左上の座標値で、値はリストで与えることができます。

```
tell application "DragEx"
    set location of Component 1 of Frame 1 to {100, 100}
end tell
```

また、Component を継承したクラスもスクリプトで指定できます。たとえば、テキストフィールドは Java では TextField ですが、用語説明にあるように AppleScript の世界では、「Text Field」と単語間にスペースが入ります。「Text Field 1 of Drag Ex Window 1」がテキストフィールドを参照するという具合です。

TextComponent では、text content というプロパティがあり、それがコンポーネント内のテキストを示します。以下のようなプログラムで、テキストフィールドに「AppleScript and Java」という文字を設定できます。

```
tell application "DragEx"
    set text content of Text Field 1 of Drag Ex Window 1 to "AppleScript and Java"
end tell
```

プログラム内で定義している DraggableLabel も、もちろん、AppleScript で参照できますが、「Draggable Label 1」のように、大文字の前を空白で区切った名前になります。たとえば、次のようなプログラムで、1 つの DraggableLabel に表示されている文字列

を「Java Object!」に変更できます。

```
tell application "DragEx"
    set text content of Draggable Label 1 of Drag Ex Window 1 to " Java Object!"
end tell
```

♦AWT のコンポーネントとして認識される

以上のように、ウインドウなどに表示した AWT のコンポーネントは、表示しただけでスクリプト制御が可能な状態になっています。一方、生成すればどのようなオブジェクトでもスクリプトプログラムで記述できるわけではないようです。たとえば、単に new で生成したオブジェクトや、static なだけのクラスは、上記のような規則で記述しても認識されません。

つまり、MRJ Scripting で認識できるオブジェクトは、AWT のビジュアル階層に組み込まれたものだけのようです。表示されるウインドウはデスクトップに追加されたものと見ることができますし、ウインドウ内のコンポーネントは add メソッドで明白に追加されたものです。こうしたAWTの階層構造に組み入れられたものが、AppleScript のオブジェクトとして認識されるようです。

認識されないクラスの用語を取り出す

Java のアプリケーション内で定義されているクラスであれば、何でもその用語を取り出すことができます。すでに AWT の管理下でインスタンス化されたクラスについてはこれまで見てきたように、自動的に用語説明に加わります。そうではないクラスについても、add terminology という AppleScript のコマンドによって、用語説明に組み入れることができます。

```
tell application "DragEx"
    add terminology for class "AppCore"
    add terminology for class "InetUtils"
end tell
```

サンプルプログラムで以上のプログラムを実行した後、再度、スクリプト編集プログラムで用語説明を開きます。すると、AppeCore と InetUtils のそれぞれのクラスの用語が加わっています。

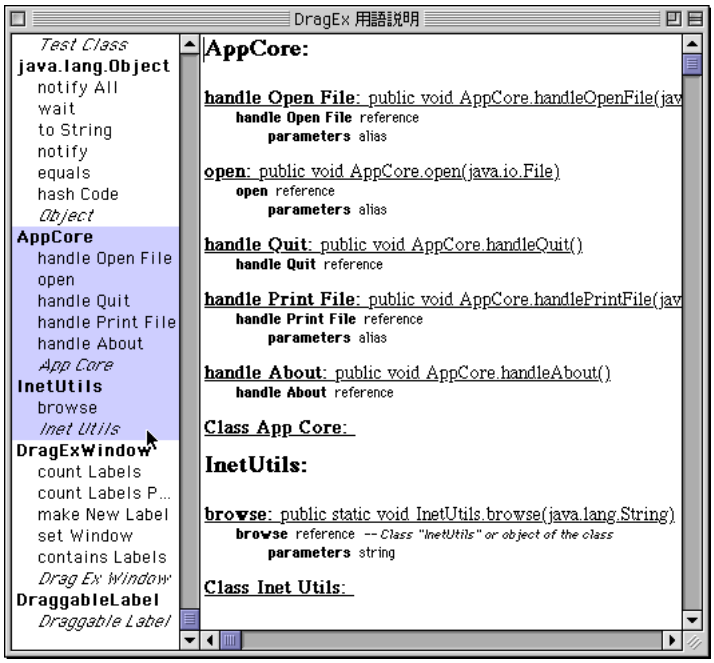

図 10.6　用語が追加された

ただし、用語が追加されたからと言って、それがスクリプトプログラムで使えるとは限りません。たとえば、サンプルプログラムの場合、AppCore クラスや TextClass クラスはインスタンスを生成しておいても、AppleScript 側からそのインスタンスをオブジェクトとして参照する方法がありません。いろいろな記述を試してみましたが、エラーなく動作することはありませんでした。MRJ Scripting による Java オブジェクト利用では、どうしてもターゲットとなるオブジェクトの指定がプロパティはもちろん、コマンドでも必要になります。AWT のオブジェクト階層に含まれないと、AppleScript でのオブジェクトとして利用できないので、単に生成したものは参照する方法がないということです。

上記の用語説明を見ると、static なクラス（InetUtils クラス）は、「Class “InetUtils”」で参照できるようです。ただし、「browse Class “InetUtils” paramteres “…”」のように、コマンドにこの記述を適用してもエラーが出ます。ただ、以下のように、Java のメソッドを呼ぶという apply to コマンドを利用することで、static なクラスのメソッドは呼び出すことができます。

```
tell application "DragEx"
    apply to Class "InetUtils" java method "browse" parameters "http://msyk.locus.co.jp/mjr/"
end tell
```

♦用語説明の情報をプログラムで生成する

BeanInfo の機能を利用して、用語説明に表示する内容を自動的に生成することも可能なようです。方法については、Appleから配付されているサンプルプログラムのMagic Oracle を参照してください。

♦取り出した用語の保存

Java のアプリケーションに対して add terminology コマンドを実行すると、そのアプリケーションの中で用語説明が追加されます。そのような方法だと、毎回 add terminology コマンドを実行しなければならなくなりますが、構築した用語説明を保存する手段も用意されています。

そのためのコマンドが、get terminologies コマンドで、aete リソースの内容をすべて取り込みます。Technote 1162 によれば、スクリプト機能拡張の GTQ Scripting Library にある Add Resource を使って、get terminologies で取得した aete リソースをファイルに保存できるとなっています。そして、保存したファイルから DeRez などを使ってリソース定義ファイルを作り、それを編集して実際に使う aete リソースを自分で編集するとなっています。

ただし、get terminologies で得られる形式と、Add Resource コマンドが認識するリソースの形式が一致していないようで、上記の方法ではエラーが出て用語説明のデータを保存することができません。これについては、MRJ のバージョンアップで、現在の用語説明の内容をファイルに保存するようなコマンドにグレードアップされるなどを待つしかないと思われます。

♦用語を取り出せなくする

自動的に用語説明が増える機能を止めることもできます。そのためには、scsz リソースで記述を行うことになっているのですが、その具体的な記述方法についてはまだ公開されていません。用語説明を取り込んで意図する形式に編集し、その結果を利用者によって変更されないようにしたいような場合には、scsz リソースを使えばいいということになります。ただ、そうしたニーズが出るところまで環境は整っていないというところでしょうか。

10-3 Java のソースと AppleScript の関係

ここまでに説明した MRJ Scripting の機能を踏まえて、Java のプログラムとスクリプトプログラムでの関連性を詳しく見ていくことにします。自分で作成したアプリケーションが AppleScript 側でどう見えるのか、あるいは意図した見え方を AppleScript 側でできるようにするために、どのように Java のプログラムを作るかと言うことの検討材料を説明しましょう。

クラスの定義

すでに説明したように、Java で定義したクラスはそのまま AppleScript でのスイートとクラスに対応付けられます。ただし、基本的には AWT でのクラスを利用することになるので、たとえば、文書内に配置するオブジェクトを Component を継承したものにするなど、AWT を中心としたクラス設計が必要になるでしょう。もっとも、Java で GUI がからむようなプログラムでは一般には AWT を使うことになるかと思われます。なお、AppleScript から利用できるクラスは public として定義されたものに限られます。

AWT 以外では static なクラスは AppleScript から利用できますが、前述のように apply to コマンドで呼び出す形式なので、プログラム的には AppleScript 的なやり方ではなく、その場しのぎ的なメソッド呼び出しです。その意味では、static なクラスのメソッド呼び出しは「できなくもない」程度のプライオリティということになるでしょう。

プロパティを定義する

クラスのプロパティは、Java のクラスでは、set あるいは get で始まるメソッドが対応します。クラスのメンバー変数が対応するわけではありません。また、is で始まるメソッドも同様に、プロパティが対応します。メソッド名から set あるいは get、is を除いた名前がプロパティになります。たとえば、次のようなクラス定義があったとします。

```
public class DragExWindow extends Frame
{
    ….

    public java.awt.TextField textField1;      //テキストフィールド
    public int countLabel = 0;                          //メンバ変数を定義する
```

```
        //メンバ変数は public でも AppleScript 側からは見えない

    /*      これらにより、text Message プロパティが定義される    */
    public void setTextMessage(String msg)        {
        textField1.setText(msg);
    }

    public String getTextMessage()    {
        return textField1.getText();
    }

    /*これにより、read only の Window プロパティが定義される   */
    public boolean isWindow()   {
        return true;
    }

    ….
}
```

まず、AppleScript 側から見えるためには、メソッドは public で宣言されている必要があります。そして、setTextMessage や getTextMessage が定義されていることで、DragExWindow による「Drag Ex Window クラス」において「text Message」プロパティが存在するように機能します。プロパティなので、「text Message of Drag Ex Window 1」に対して代入あるいはこの形式で参照できるというわけです。このとき、メソッド名の大文字の前に空白を入れた形式の単語に別れた文字列が AppleScript 側では対応しています。

ここで、set あるいは get で始まるメソッドだけが定義されていても、やはりプロパティとして認識されます。get だけならリードオンリーになります。

is で始まるメソッドも同様にプロパティとして認識され、前の例のようなメソッドが定義されていれば、is を抜かした window という論理型でリードオンリーのプロパティがあるというような動作をします。なお、is で始まるメソッドがある場合、同じプロパティに対応する set や get で名前が始まるメソッドがあっても、それらはプロパティにはならず、次に説明するコマンドとして認識されます。

前のプログラムをアプリケーション化して実行し、アプリケーションの用語説明を開くと次のようになっています。Drag Ex Window クラスのプロパティが、Java のメソッドに対応しています。

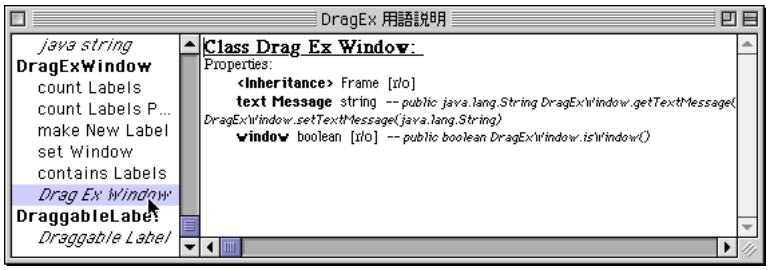

図 10.7　**メソッドがプロパティとして認識されている**

コマンドを定義する

プロパティとして認識する以外のメソッドは、すべてコマンドとして認識されます。コマンドも、public なメソッドであることが前提ですが、メソッド名がそのままコマンド名になります。他のものと同様、メソッド名は大文字の前で空白が入ったような単語に別れた名前のコマンドになります。たとえば、次のようなクラス定義があったとします。その場合のコマンド一覧を用語説明で見てみます。

```
public class DragExWindow extends Frame
{
    …

    public int countLabels()        {
            System.out.println("Call: countLabels, no parameters");
            return countLabel;
    }

    public int countLabels(int param)   {
            System.out.println("Call: countLabels, 1 parameters");
            return countLabel;
    }

    public int countLabels(int param1, String param2)    {
            System.out.println("Call: countLabels, 2 parameters");
            return countLabel;
    }
    /*      オーバーロードは正しく処理される。parameter 後の指定で、呼び出すメソッドを
            選択できる。ただし、用語集には正しくは反映されない。        */

    public int countLabelsParam(int param1, String param2)  {
            System.out.println("Call: CountLabelsParam");
            return countLabel;
    }

    public int containsLabels(int param1, String param2)        {
            System.out.println("Call: containsLabels");
            return countLabel;
    }
```

```
    /*      メソッド名を変えた時に注意をする。
            特にデバッグ中。
            スクリプト編集プログラムのウインドウはいったん閉じた方がいい  */

    private void setupMenuBar() //private なので AppleScript 側からは見えない
    {

    …

    }

    /*      オブジェクトを生成するメソッド。コンストラクタは AppleScript 側からは
            見えないので、生成が必要ならこうしたメソッドを用意する。
            戻り値として作成したオブジェクトの参照を戻しているが、AppleScript 側では
            それはうまく扱えない模様   */
    public DraggableLabel makeNewLabel(String text)  {
            DraggableLabel newLabel;
            add(newLabel = new DraggableLabel(text,200,250,150,50,Color.orange));
            return newLabel;
    }
}
```

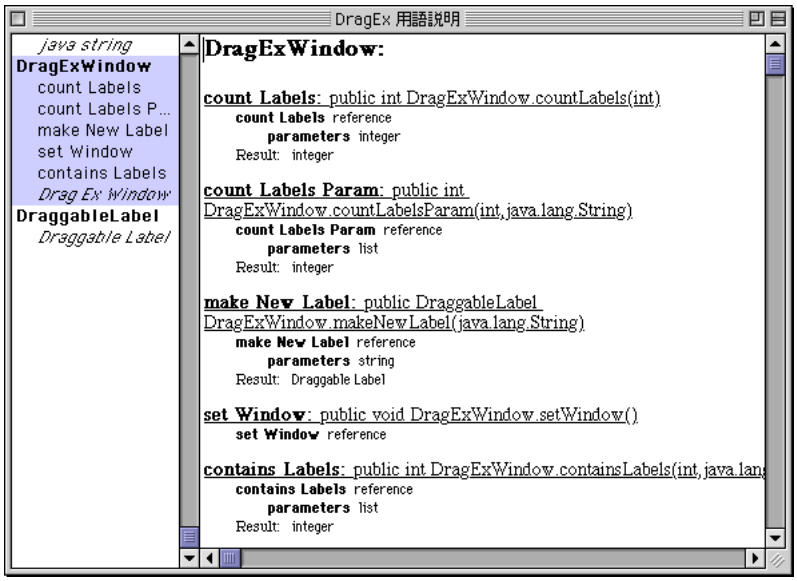

図 10.8　**メソッドがコマンドとして認識されている**

たとえば、countLabels メソッドは、「count Labels コマンド」になっているのが 1 つのポイントです。そして、どのコマンドも、ダイレクトパラメータに、適用するオブジェクトを指定することになります。たとえば「count Labels Drag Ex Window 1」というので、表示されているウインドウのクラスの countLabels メソッドを呼び出します。

メソッドに引数がある場合、一様に、parameters に続いて順序に従って指定します。このとき、実行時には引数の型をしっかりチェックしているので、メソッドの指定通

りに parameters 以降を指定します。2 つ以上の引数がある場合、引数はリストの形式で指定してやる必要があります。

オーバーロードしたメソッドも正しく認識できます。AppleScript 側での parameters 以降の指定によって、複数ある同一名称のメソッドのいずれかを選択して呼び出します。ただし、オーバーロードしたメソッドについては、そのことが分かるような用語説明にはなっていません。最終的には aete リソースを得て内容を書き直すなどの処置が必要になるところです。

♦デバッグ中の注意

プログラムを作成しながら、スクリプト編集プログラムで、AppleScript によるテストを行うようなこともあるかも知れません。そのとき、プログラムのメソッド名など、AppleScript 側の定義に変化が出るような修正を行った場合、いったん、スクリプト編集プログラム側で開いているウインドウを閉じ、改めて開くなどしてください。そいうしないと、開いているウインドウが、古い状態のアプリケーションの情報を保持したままになり、プログラムの修正結果などが反映されず、思わぬエラーに見舞われることもあります。ウインドウを開き直して、アプリケーション情報の更新を行うことがいちばん確実です。

オブジェクトを生成する

クラスのコンストラクタは AppleScript 側には公開されないようです。そのため、AppleScript 側からインスタンスの生成をしたいような場合には、そのためのメソッドを作成することになります。単に生成したオブジェクトが AppleScript では認識されないので、コンストラクタだけの呼び出しが AppleScript から出来たとしてもあまり意味はないと言うことになるでしょう。たとえば、次のようなメソッドを定義しておくと、

```
public class DragExWindow extends Frame
{
    …

    /*    オブジェクトを生成するメソッド。コンストラクタは AppleScript 側からは
          見えないので、生成が必要ならこうしたメソッドを用意する。
          戻り値として作成したオブジェクトの参照を戻しているが、AppleScript 側では
          それはうまく扱えない模様   */
    public DraggableLabel makeNewLabel(String text)  {
          DraggableLabel newLabel;
          add(newLabel = new DraggableLabel(text,200,250,150,50,Color.orange));
          return newLabel;
    }
```

```
    …
}
```

このメソッドにより、AppleScript 側では Drag Ex Window スイートで、「make New Label」というコマンドが使えるようになります。「make New Label Drag Ex Window 1」のようなプログラムで、上記のメソッドが呼び出されます。メソッドでは、DraggableLabel のコンストラクタが呼び出されてさらに、add メソッドによりそれがウインドウに追加されています。そして、生成したオブジェクトへの参照を戻します。戻り値は AppleScript 側で変数に代入できなくもないのですが、その結果に対してプロパティなどを適用してもうまく動きません。やや残念なところです。

メソッドを呼び出す

一般的な方法、つまり用語説明のクラスで表示されたプロパティや、スイートに含まれるコマンドを使うことで、メソッドの呼び出しができますが、直接メソッド呼び出しが可能なコマンドが、Java のアプリケーションでは使えるようになっています。以下のようなコマンドが AppleScript で使えます。

apply to オブジェクト java method "メソッド名" parameters 引数

これによりオブジェクトに定義されているメソッド名のメソッドを呼び出します。引数も単一の値あるいはリストで複数の値を指定します。オブジェクトでは、AWT の階層に組み入れられているものだけではなく、「Class "InetUtils"」のように static なクラスを直接指定することが出来ます。その場合、クラス名の文字列は、Java の側で定義されたクラス名を指定します。

アプリケーションへのアクション

Java のアプリケーションへのキータイプやマウスクリックを、AppleScript から発生させるコマンドが利用できます。これも、Java のアプリケーションであれば、使えるコマンドです。

以下の AppleScript コマンドは、「対象オブジェクト」に指定したウインドウに対して文字列をキータイプします。対象オブジェクトを省略するとアクティブなウィンドウに対してキータイプ処理を行います。最後の整数は、コマンドキーや Control キーなどのモディファイアキーを指すものだと思われますが、どの数値がどのキーに対応するのかと言う情報は得られませんでした。

type into [対象オブジェクト] keystrokes 文字列 [with modifiers 整数]

以下の AppleScript コマンドは、「対象オブジェクト」に指定した AWT のコンポーネントをクリックします。座標値を示すわけではないので、つまりは指定したオブジェクトにアクションイベントを発生させるものと思われます。最後の selecting パラメータは選択肢のあるオブジェクトで特定の項目を選択するための指定ですが、あとのパラメータについては詳細は不明です。

click 対象オブジェクト [at 整数] [with modifiers 整数] [selecting null/some Java object]

♦スクリプトから行えるその他の操作

オブジェクトのクラスをチェックする check コマンドがありますが、いろいろやった限りでは何を指定しても false になってしまいました。

オブジェクトのシリアライズを行ってファイルに保存する save object や、逆にファイルからオブジェクトを生成する load object というコマンドも利用できます。

Java Bean ベースのツールを開始する start tool というコマンドもあります。

以上については、用語説明を参照してください。

10-4 サンプルプログラム

この章で紹介したアプリケーションのソースを示しておきます。また、このアプリケーションをコントロールする AppleScript のプログラム例も紹介しておきましょう。

AppCore.java

アプリケーションに使うソースファイルは 4 つで、他にリソースファイルを 2 つプロジェクトに登録しています。1 つは MRJ SDK に付属する MRJ Scripting aete で、もう 1 つは BNDL リソースやアイコン情報を記録したリソースです。

以下の、AppCore は一般的な Mac OS での Java アプリケーションのいちばん中心になる部分で、必須イベントの処理を行っています。ただし、ファイルの読み込み等の具体的な機能は組み込んでいません。この AppCore クラスのインスタンスが AppleScript から認識できるかをチェックすることが大きな目的です。また、内部で定義している TextClass クラスが同様に AppleScript から認識できるかをチェックしています。たとえば、スクリプトプログラム中の「App Core 1」「Test Class 1」のような記述が可能なのかを調べてみましたが、結論としては認識できないということになります。

```
import com.apple.mrj.*;
import java.io.*;
import java.util.*;
import java.awt.*;
import java.awt.event.*;

/*   このクラスを用語集に表示させるには、add terminology コマンドを使う   */

public class AppCore
    implements      MRJOpenDocumentHandler,
                    MRJPrintDocumentHandler,
                    MRJQuitHandler,
                    MRJAboutHandler
{
    static public void main(String arg[])      //起動時に実行されるメソッド
    {
        new AppCore();  //このクラスを新たに生成する
    }

    public AppCore() //コンストラクタ
```

```
    {
        /*基本 AppleEvent が、このクラスに伝達されるように定義する*/
        MRJApplicationUtils.registerOpenDocumentHandler(this);
        MRJApplicationUtils.registerPrintDocumentHandler(this);
        MRJApplicationUtils.registerQuitHandler(this);
        MRJApplicationUtils.registerAboutHandler(this);

        new DragExWindow();
        keep = new TestClass();
    }

    TestClass keep;

    public void handleOpenFile(File filename)      //Open イベントが Finder からやってきたとき
    {
        System.out.println("Dragged "+filename.toString());
    }

    public void handlePrintFile(File filename)      //Print イベントが Finder からやってきたとき
    {
        System.out.println("Print Event "+filename.toString());
    }

    public void handleQuit()      //Quit イベントが Finder からやってきたとき、
                                  //あるいはアップルメニューの「終了」を選択したとき
    {
        System.out.println("Quit Event Received");
        System.exit(0);   //アプリケーションの終了
    }

    public void handleAbout()    //アップルメニューの About を選択したとき
    {
        System.out.println("Select About Menu");
    }

    public void open(File filename)     //Open イベントが Finder からやってきたとき
    {
        System.out.println("Opening "+filename.toString());
    }

}

/*  このクラスを用語集に表示させるには、add terminology コマンドを使う。
    インスタンス化しなくても、用語集を取り出すことができる。
    ただし、オブジェクト自体は AppleScript 側からは参照できない              */

public class TestClass      {
    public String testMethod(int param)      {
        return "This is Test Method";
    }
}
```

DragExWindow.java

アプリケーションのウインドウを管理するクラスです。プロパティやあるいはコマンドが実際にどうメソッドと関連するのかを見るために、いろいろなパターンのメソッドを定義してみました。詳細については、10-2 や 10-3 で説明しています。

```
import java.awt.*;
import java.awt.event.*;

public class DragExWindow extends Frame
{
    public DragExWindow()
    {
        setLayout(null);  //レイアウト機能は使わない
        setSize(500,365);       //ウインドウのサイズを指定する

        textField1 = new java.awt.TextField();    //テキストフィールドを配置する
        textField1.setBounds(12,10,480,21);     //テキストフィールドの位置と大きさ
        add(textField1);  //テキストフィールドをウインドウに追加する

//          add(new DraggableLabel("ドラッグしてみよう",40,40,120,30,Color.red));
//          add(new DraggableLabel("これもドラッグする",200,250,150,50,Color.orange));

        setupMenuBar(); //メニューを構築する
        show();                 //ウインドウを表示する
    }

    public java.awt.TextField textField1;      //テキストフィールド
    public int countLabel = 0;                          //メンバ変数を定義する
        //メンバ変数は public でも AppleScript 側からは見えない

    /*      これらにより、text Message プロパティが定義される    */
    public void setTextMessage(String msg)        {
        textField1.setText(msg);
    }

    public String getTextMessage()    {
        return textField1.getText();
    }

    /*これにより、read only の Window プロパティが定義される   */
    public boolean isWindow()   {
        return true;
    }

    /*      これは Window プロパティがすでにあるために、コマンドになる     */
    public void setWindow()       {

    }
```

```
public int countLabels()        {
        System.out.println("Call: countLabels, no parameters");
        return countLabel;
}

public int countLabels(int param)   {
        System.out.println("Call: countLabels, 1 parameters");
        return countLabel;
}

public int countLabels(int param1, String param2)    {
        System.out.println("Call: countLabels, 2 parameters");
        return countLabel;
}
/*      オーバーロードは正しく処理される。parameter 後の指定で、呼び出すメソッドを
        選択できる。ただし、用語集には正しくは反映されない。        */

public int countLabelsParam(int param1, String param2)  {
        System.out.println("Call: CountLabelsParam");
        return countLabel;
}

public int containsLabels(int param1, String param2)        {
        System.out.println("Call: containsLabels");
        return countLabel;
}

/*      メソッド名を変えた時に注意をする。
        特にデバッグ中。
        スクリプト編集プログラムのウインドウはいったん閉じた方がいい  */

private void setupMenuBar()
{
        MenuItem newItem, openItem, closeItem, quitItem;

        MenuBar mb = new MenuBar();

        Menu fileMenu = new Menu("メニュー", true);
        newItem = new MenuItem("ラベル作成", new MenuShortcut('N'));
        newItem.addActionListener(new NewItemListener());
        fileMenu.add(newItem);

        fileMenu.addSeparator();

        quitItem = new MenuItem("終了", new MenuShortcut('Q'));
        quitItem.addActionListener(new QuitItemListener());
        fileMenu.add(quitItem);
        mb.add(fileMenu);

        setMenuBar(mb);
}

/*      オブジェクトを生成するメソッド。コンストラクタは AppleScript 側からは
        見えないので、生成が必要ならこうしたメソッドを用意する。
```

```
        戻り値として作成したオブジェクトの参照を戻しているが、AppleScript 側では
        それはうまく扱えない模様  */
    public DraggableLabel makeNewLabel(String text)  {
        DraggableLabel newLabel;
        add(newLabel = new DraggableLabel(text,200,250,150,50,Color.orange));
        return newLabel;
    }

    /*    メニューを選択たときに、ラベルを追加する処理   */
    class NewItemListener implements ActionListener   {
        public void actionPerformed(ActionEvent ev)  {
            add(new DraggableLabel("これもドラッグする",200,250,150,50,Color.orange));
            countLabel++;
        }
    }

    class QuitItemListener implements ActionListener   {
        public void actionPerformed(ActionEvent ev)  {
            System.exit(0);
        }
    }
}
```

DraggableLabel.java
ウインドウ内でドラッグが可能なラベルの定義です。10-2 で説明したように、AppleScript でコントロールできることを確認できますが、むしろドラッグ可能にする方法のサンプルとしての価値の方が高いかも知れません。

```
import java.awt.*;
import java.awt.event.*;

/*   ドラッグ可能なラベルの定義                */

public class DraggableLabel extends java.awt.Label
    implements MouseListener,MouseMotionListener
{
    int biasX, biasY;
    int eventIgnore=0;

    public DraggableLabel(String label,int x, int y, int width, int height,Color col)
    {
        super(label);       //親クラスのコンストラクタの呼び出し
        setBounds(x,y,width,height); //引数に応じてラベルの設定を行う
        setBackground(col);
        this.addMouseMotionListener(this);       //リスナーを追加する
        this.addMouseListener(this);
    }

    public void mouseDragged(MouseEvent aEvent)    {       //ドラッグした時に呼び出される
        if(eventIgnore>0)       {
            this.setLocation( this.getLocation().x+aEvent.getX()-biasX,
                                this.getLocation().y+aEvent.getY()-biasY);
                //マウスポインタの現在の位置に、ラベルを移動する
            eventIgnore=0;
```

```
            }
            else
                eventIgnore++;
                    //この変数により、ドラッグの動作をスムーズにするようにしている
                    /*ドラッグを検知して、ラベルの位置を setLocation で移動すると、
                    そのメソッド呼び出しがさらに mouseDragged を呼び出すことになって
                    しまう。そこで、setLocation 後の最初の mouseDragged の呼び出しを
                    キャンセルするために、eventIgnore 変数を使う。こうしないと、
                    ラベルがあっちこっちに飛び回るような表示になってしまう。*/
        }

        public void mouseEntered(MouseEvent aEvent)             //マウスがラベル内に入った時
        {
            setCursor(new Cursor(Cursor.HAND_CURSOR));  //マウスポインタを手の形に変更
        }

        public void mousePressed(MouseEvent aEvent)     //ドラッグ開始時に呼び出される
        {
            biasX = aEvent.getX();//マウスの位置をラベルの左上からの座標値として記録
            biasY = aEvent.getY();
        }

        public void mouseMoved(MouseEvent aEvent)       {       }       //空定義が必要
        public void mouseClicked(MouseEvent aEvent)     {       }       //空定義が必要
        public void mouseExited(MouseEvent aEvent)      {       }       //空定義が必要
        public void mouseReleased(MouseEvent aEvent)    {       }       //空定義が必要
    }
```

InetUtils.java

static なメソッドが AppleScript からどうすれば認識できるかを試すためのものです。10-2 で示したように、apply to を用いれば、呼び出しが可能です。ただし、実際の処理は、AppleEvent を送付しているので、AppleScript から呼び出す価値はまったくないと言えるでしょう。

```
import com.apple.MacOS.*;
import java.io.*;
import java.lang.Runtime;

public class InetUtils {

static public void browse(String urlString)     {

    String osName = System.getProperty("os.name");
    if (osName.indexOf("Mac OS") >=0)               {
        AEDesc directParam = new AEDesc(urlString);
                            //開くアドレスの文字列をディスクリプタ化する
        AEDesc errRepParam = new AEDesc(false);
                //パラメータの error reporting に指定する論理値をディスクリプタ化する
```

```
            AEDesc finderAddress = new AEDesc(
                        //イベントの送付先は Finder にする。ただし open location は OSAX
                        //の機能なので、このイベントは何に送ってもかまわない
                    AEDesc.typeApplSignature,
                        //      送付先をクリエイタで指定することを示すタイプを設定
                    OSUtils.makeOSType("MACS"));        //      Finder のクリエイタ
            AETarget finderTarget = new AETarget(finderAddress);
                        //イベントの送付先を示すオブジェクトを生成
            AppleEvent sendingEvent = new AppleEvent( //Apple イベントを作成する
                    OSUtils.makeOSType("GURL"),         //      イベントクラスを指定
                    OSUtils.makeOSType("GURL"),         //      イベント ID を指定
                    finderTarget);                      //      送付先を指定
            sendingEvent.putParameter(                  //ダイレクトパラメータをイベントに設定
                    AEDesc.keyDirectObject,             //      ダイレクトパラメータを示す値を指定
                    directParam);                       //      パラメータに設定する値
            sendingEvent.putParameter(          //error reporting パラメータをイベントに設定
                    OSUtils.makeOSType("errr"),     //      ダイレクトパラメータを示す値を指定
                    errRepParam);                   //      パラメータに設定する値
            sendingEvent.sendNoReply();         //イベントを送付する。応答はなし
        }
        else  {
            try    {
                Runtime.getRuntime().exec("rundll32 url.dll,FileProtocolHandler "+urlString);
                    //Java ワールド、99/6 号、p142-143
            }
            catch(Exception e)     {       System.out.println(e.getMessage());      }
        }
    }
}
```

TestScript1

Java のアプリケーションをそれなりにコントロールできることを示すサンプルです。アプリケーションの DragEx を起動した状態で以下のスクリプトを実行してください。すると、新たにラベルが 3 つ作成され、存在するコンポーネントの背景色をランダムに設定します。背景色は Component の background プロパティで設定できますが、値は RGB 値をそれぞれ要素に持つリストを指定すればいいようです。各要素は 0〜255 の値になればいいようです。そして、100 回のループの中で、各コンポーネントの座標位置をランダムに移動しながら、表示テキストを変更しています。テキストフィールドには現在時刻が入力されるようにしてみました。

このスクリプトを稼働中に、DragEx アプリケーションの中のラベルをドラッグしてみてください。スクリプト実行中でも Java のアプリケーションは操作を受け付けることが分かります。

```
tell application "DragEx"
	activate

	make New Label Drag Ex Window 1 parameters "新規作成ラベル 1"
	make New Label Drag Ex Window 1 parameters "新規作成ラベル 2"
	make New Label Drag Ex Window 1 parameters "新規作成ラベル 3"

	set cNum to component Count of Drag Ex Window 1

	repeat with i from 1 to cNum
		set r to random number from 150 to 255
		set g to random number from 150 to 255
		set b to random number from 150 to 255
		set background of Component i of Drag Ex Window 1 to {r, g, b}
	end repeat

	repeat 100 times
		repeat with i from 1 to cNum
			set x to item 1 of location of Component i of Drag Ex Window 1
			set y to item 2 of location of Component i of Drag Ex Window 1
			set x to x + (random number from -50 to 50)
			set y to y + (random number from -50 to 50)
			if x < 0 then set x to 0
			if y < 0 then set y to 0
			set locList to {x, y}
			set (location of Component i of Drag Ex Window 1) to locList

			set (text content of Component i of Drag Ex Window 1) to "AppleScript でコントロ
ール中"

			set textMessage of Window 1 to "現在の時刻：" & (current date)
		end repeat
	end repeat
end tell
```

TestScript2.java

このサンプルは、apply to の利用例です。ウインドウのタイトルを、Frame クラスの setTitle メソッドを呼び出して指定しています。

```
tell application "DragEx"
	set titleString to "AppleScript で設定したタイトル：" & (current date)
	apply to (Drag Ex Window 1) java method "setTitle" parameters titleString
	--	apply to (Drag Ex Window 1) java method "setTitle" parameters ""
end tell
```

TestScript3.java

このサンプルは、用語説明のデータを取り込んでファイルに aete リソースとして保存することを目的としたものですが、10-2 に説明したように、エラーが出て機能しませんでした。

```
tell application "DragEx"
	add terminology for class "DraggableLabel"
	set aeteData to get terminologies
end tell
add resource (aeteData as resource) to (file "partner:Test.rsrc") id 0 with replacing allowed
```

TestScript4.java

10-2 でサンプルとして示したスクリプトプログラムをまとめておきました。

```
tell application "DragEx"
	add terminology for class "DraggableLabel"
	add terminology for class "AppCore"
	add terminology for class "InetUtils"
end tell
tell application "DragEx"
	set title of Drag Ex Window 1 to "AppleScript and Java"
	set location of Component 1 of Frame 1 to {100, 100}
	set text content of Text Field 1 of Drag Ex Window 1 to "AppleScript and Java2"
	set text content of Draggable Label 1 of Drag Ex Window 1 to " Java Object!"
end tell
```

（第 10 章以上）

第11章 QuickTime for Java の利用

ビデオ画像やサウンドなどの時間軸上に展開されるデータを中心に、マルチメディアにかかわるあらゆるデータを扱う基本システム機能がQuickTime です。

その QuickTime を使ったプログラムを、Java でも制作できるようになりました。QuickTime for Java という機能を組み込むことで、QuickTime の諸機能が Java のライブラリという形式で利用できるようになります。

一般的なアプリケーションでムービーを表示し、ある程度のコントロールを行うというあたりの基本的な利用について、この章では説明します。

11-1 QuickTime for Java を使う

Java と QuickTime を結びつけるのが QuickTime for Java です。QuickTime for Java によって、Java 開発環境あるいは実行環境がどう変わるのかについて、概略を説明しましょう。

この章で説明する範囲

QuickTime の機能は非常に豊富で、ビデオの画像取込みやさまざまな編集にまでびますが、この章では、QuickTime for Java を使ってムービーを表示し、簡単な編集を行うところまでを説明します。一般的なアプリケーションでは、ムービーを表示するということでおおむね事が足りると考えました。

QuickTime for Java にはドキュメントがしっかり用意されており、書籍も発行されています。この章の範囲を超える詳細を知りたい場合はそちらを御覧ください。

QuickTime と Java

Java の歴史はまだまだ浅いのですが、すでにそれなりにライブラリが整備されて、アプリケーションを作るような場合でも、機能的にそん色があまりないレベルにまで来ています。しかしながら、マルチメディアの分野はいまだに弱いと言えるでしょう。Java Media Framework という Java のライブラリが登場しているものの、これには機能的な制約があり、また、クロスプラットフォーム仕様であるとはいえ、実際に使えるのは Windows と Solaris だけで、AWT などのようなコアなライブラリとは違って Mac OS 環境では使えません。

そのような中、98 年 3 月に Apple は、QuickTime の機能を Java で利用できるようにする QuickTime for Java の開発を発表しました。ただし、この時点では、機密保持条項に同意した人にだけ配付されるというもので、「開発している」こと以外はほとんどが公にできないくらいでした。QuickTime for Java のベータ版が配付されはじめたのはその 1 年後の 99 年 3 月で、その時点でやっと開発者以外が QuickTime for Java を使えるようになったと言えるでしょう。

98 年 3 月の時点では、Java Media に対する評価もまだこれからという時期だったこともあって、すでにシステムに搭載されていて、Apple のマルチメディア分野のアド

バンテージである QuickTime を利用した QuickTime for Java は、大きく取り上げられました。QuickTime は Mac OS では標準搭載されていますが、Windows 版もあることから、事実上 QuickTime for Java はマルチプラットフォームであるとも言えるわけです。

ただ、ベータ版が出てくるまでの 1 年で、ある意味では状況が変わってきました。Windows 環境では、Microsoft 社がそれなりにマルチメディア機能を高めた結果、般のユーザーが、QuickTime を組み込まなくてもムービーを見ることはできるような環境が整ってきました。そのため、Windows 版の QuickTime は、一般的な使用目的ではあえてインストールする必要がなくなっています。QuickTime がシステムにセットアップされていないといけない QuickTime for Java は、結果的に機能的にはクロスプラットフォームではあるものの、実質的には Mac OS での利用以外はスムーズではないということになってしまっています。たとえば、Web ページで利用するアプレットで QuickTime for Java の機能を使った場合、そのページを開いてアプレットが正しく機能するユーザーは、Mac OS のユーザーと言うことになり、多くの Windows ユーザーは、QuickTime のインストールなどをしないといけないという結果になるわけです。

このように考えると、QuickTime for Java は、Mac OS 向けのアプリケーション開発において、QuickTime の機能を気軽に効果的に使えるライブラリとしての位置付けがもっとも中心的になるものと思われます。C 言語などのアプリケーション作成では、QuickTime の API を通じて機能を手軽に利用できますが、Java の標準ライブラリではマルチメディアデータの扱いは非常に制約されています。そのような中、QuickTime を使えるメリットは、アプリケーションの中でムービーデータを気軽に使えるということに他なりません。Mac OS なら QuickTime がシステムに標準で組み込まれているので、動作環境としてのハードルも低いでしょう。また、そうしたアプリケーションは、Windows でも使えますが、ただし、こちらは動作環境のハードルがいくらか高くなることは計算に入れておく必要があるでしょう。

利用するクライアント環境

QuickTime for Java を使ったプログラムを実行する環境について説明をしておきましょう。つまり、実行可能な環境についてということになります。

まず、基本的なことですが、QuickTime for Java を利用するには、QuickTime がシステムに組み込まれている必要があります。QuickTime for Java があるだけでいいというわけではなく、実行するクライアントに必ず QuickTime が組み込まれている必要があ

ります。アプリケーションなら実行するマシンのシステムに QuickTime が組み込まれている必要があります。アプレットなら、ダウンロードして実行するクライアントに QuickTime が組み込まれている必要があります。QuickTime は、Ver.3 以降のものが必要です。

そして、Java の実行環境が存在する必要があります。Mac OS だと、MRJ 2.1 以降の Java 実行環境が必要です。Windows の場合は、JDK 1.1.x のものが利用できますが、現状では、1.1.8 を利用することになるでしょう。Java2 環境にも対応しています。

こうした土台の上に、QuickTime for Java をインストールすることになります。Mac OS 向けの QuickTime for Java は、「機能拡張」フォルダにある MRJ Libraries フォルダの中の MRJ Classes フォルダに、QTJava.zip というアーカイブされたライブラリをインストールするのみです。つまり、この QTJava.zip というたくさんのクラスファイルをアーカイブしたファイルが、QuickTime for Java の実体です。つまり、QuickTime for Java は、QuickTime というネイティブ機能を使う Java のライブラリであるというわけです。

♦アプレットの実行環境

QuickTime for Java を利用したアプレットを作成し、それをネットワーク経由でダウンロードする一般的なアプレットとして利用することも可能です。その場合でも、基本的には、クライアント側で、MRJ あるいは JRE などの Java 実行環境、QuickTime、そして QuickTime for Java がセットアップされている必要があります。

たとえば、Swing などのように、QuickTime for Java のライブラリをクライアントにインストールしなくても使えないのかというニーズもあるかもしれませんが、基本的にはできるようにはなっていません。というのも、QuickTime for Java のライブラリファイルは zip フォーマットで、独自に jar フォーマットにするなどして試してみると言うことになります。もっとも、こうした形式で配付していないということは、そうした使い方を想定していないこととも考えられるので、やはり基本は QuickTime for Java をクライアントにインストールするということになっているのでしょう。

また、こうした使い方をしていても、セキュリティに引っ掛かります。ここでのセキュリティとは、アプレットではファイル処理ができないといような意味でのセキュリティです。サンプルとして提供されているアプレットも、ファイルとして開く分にはセキュリティチェックは引っ掛かりませんが、サーバーに登録すると、.class ファイルと別の画像データをファイル処理していることから、セキュリティに引っ掛かり、実行は中断されます。どのメソッドの利用が可能なのかという情報がないために、下

手をすると試行錯誤を繰り返さないといけません。

アプレットとして作成することが必須とされる場合で、QuickTime for Java を使う場合には、こうした点も留意しておく必要があります。

開発環境

QuickTime for Java を利用する Java のソフトウエアを開発する場合、通常と異なるのは、開発環境のプロジェクトに、「機能拡張」フォルダにある MRJ Libraries フォルダの中の MRJ Classes フォルダ内の QTJava.zip というファイルを追加しておくことです。

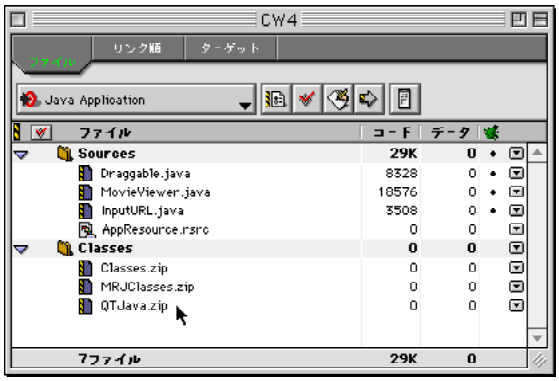

図 11.1　QTJava.zip **を加えたプロジェクト**

あとは、QuickTime for Java で利用できるさまざまなクラスを、プログラムで自由に利用することができます。

QuickTime for Java とは別に、開発者向けに The QuickTime for Java Software Development Kit（いわゆる SDK）が配付されています。この SDK には、QuickTime for Java で利用できる各クラスの JavaDoc 形式のドキュメントと、かなり豊富なサンプルプログラム集が含まれています。これらはなくても開発はできなくはありませんが、ドキュメントを見ないとどのようなメソッドがあるのか分かりませんし、サンプルは大いに参考になります。開発する人は必ずダウンロードしておくべきでしょう。

SDK をダウンロードすると、StuffIt で圧縮されたファイルが得られます。それを解凍すると、API のドキュメントは全てファイルに展開されます。ドキュメントを見るには、qtjavadocs→doc→packages.html とたどって、packages.html を Web ブラウザで開いて参照すればよいでしょう。なお、同じものは、Apple の開発者向けのサイトでも参照できます。

サンプルプログラムを解凍するには、さらに、SDK のファイルを解凍した中にあ

る Demos.sit.hqx というファイルを解凍します。すると、サンプルプログラムをインストールするインストーラが出てくるので、それをダブルクリックしてインストール作業を実際に行います。

♦アプレットの開発

アプレットも通常通り開発できますが、アプレットを呼び出すタグは APPLET だけでなく、EMBED もあるなど、環境に依存してしまいます。Apple の開発者向けの Web ページなどでもこうした情報を流していますが、実用面では SDK のサンプルプログラムのやり方をそのまま利用するのが手軽でしょう。SDK のサンプルプログラムのうち、Applets フォルダにある AppletTag.js という JavaScript のソースファイルを利用します。利用する方法の具体例は、Applets フォルダにいくつかあるサンプルを御覧になってください。

11-2 ムービーを表示する

ファイルやあるいはインターネット上のサーバーにあるムービーを、Java のアプリケーションやアプレットで表示するもっとも基本的な方法を説明しましょう。

QuickTime の使用準備

QuickTime for Java を利用する Java のアプリケーションやアプレットは、QuickTime の初期化を最初の段階で行う必要があります。そのためのメソッドが、quicktime というパッケージの QTSession というクラスに用意されています。初期化と終了のメソッドは以下の通り、いずれもクラスメソッドなので、単に呼び出しをどこかに記述しておくだけでかまいません。

表 11.1　QuickTime **の初期化と終了**

戻り値	メソッド	機能
void	QTSession.open()	QuickTime を初期化する。アプリケーションやアプレットの起動時に呼び出す。アプレットなら init メソッド内に記述する
void	QTSession.close()	QuickTime の利用を終了する。アプリケーションやアプレットの終了時に呼び出す。アプレットなら、destroy メソッド内に記述する

アプレットの場合には、Applet を extends したクラスの init と destroy メソッドに記述することでおおむね大丈夫でしょう。アプリケーションの場合には、QuickTime の処理が始まるまでの適当な段階で、open メソッドを呼びます。また、終了まじかに close を呼び出せばいいのですが、close を呼び出さないでアプリケーションを終了してもとりあえずは大丈夫なようです。ただし、こうしたメソッドが用意されているので、極力呼び出すようにすべきでしょう。

QTCanvas の利用

QuickTime のムービーはファイルあるいは URL に存在するのですが、一般には、

画像やテキストなどと異なり、必ずしもメモリーにロードしてそれを利用するとは限りません。ムービーを表示するだけなら、一般にはムービーの情報だけをまずロードして、実際の映像やサウンドのデータは QuickTime が必要に応じて自動的にロードするようになっています。こうした一般的な利用を考えます。

まず、ムービーを表示したい場合には、quicktime.app.dsiplay というパッケージにある QTCanvas というクラスを利用します。QTCanvas は、java.awt.Canvas を継承したものですので、つまりは、AWT のコンポーネントとして扱えるものです。Canvas は、たとえば JPEG イメージをウインドウやアプレット描画領域に表示することに使いました。これと同じように、QTCanvas は、QuickTime ムービーを AWT 管理下の Conainer（たとえば Frame など）に表示するのに使えるコンポーネントということになります。このように、AWT ベースのムービー表示コンポーネントが用意されているため、ムービーを表示するということはさほど難しいことではありません。基本的な機能は以下のようになっていますが、AWT の Component クラスを継承しているので、そちらの情報も合わせて検討してください。

表 11.2　QTCanvas の基本的な機能

戻り値	メソッド	機能
（コンストラクタ）	QTCanvas()	QTCanvas オブジェクトを生成する
void	setClient(Drawable, boolean)	クライアントを設定する。1 つ目の引数でクライアントを指定、2 つ目の引数が true なら現在の Canvas をムービー合わせて再レイアウトし、false なら現在の Canvas の大きさに合わせる
void	setClient(Drawable, QDRect)	クライアントを設定する。1 つ目の引数でクライアントを指定、2 つ目の引数でクライアントの位置とサイズを指定
void	setBounds(int, int, int, int)	位置と大きさを指定する。引数は順に、左上の横座標、左上の縦座標、幅、高さを指定
void	setSize(Dimension)	大きさを指定する
void	setSize(int, int)	大きさを指定する
void	setVisible(boolean)	引数が true なら表示、false なら非表示にする

ここで、setClient というメソッドがあります。「クライアント」とは、QTCanvas が

表示するムービーや画像など、QTCanvas によって管理されるものを一般的に指しています。QTCanvas によって処理される中身の事と言えばよいでしょうか。このクライアントは、Drawable という interface クラス（quicktime.app.display パッケージに含む）を指定することになっています。interface なので、実体はさらにそれを組み込んだクラスになるのですが、この Drawable という QTCanvas に表示可能な機能の事を「描画可能」というように以下で表現します。Drawable を組み込んだクラスは、QTCanvas に描画可能なオブジェクトと表現することにします。具体的に描画可能なオブジェクトを得る方法は、後で説明します。

♦ウインドウにムービーを表示する

QTCanvas にムービーや画像をクライアントとして設定すると、その画像がコンポーネント内に表示されます。引数のないコンストラクタで QTCanvas を作成した場合、QTCanvas の大きさにムービーは伸縮されます。ごく一般的なアプリケーションとして、Frame クラスを拡張したムービー表示ウインドウを定義したとしましょう。そのクラスに生成した QTCanvas オブジェクトを add メソッドで追加するとします。この QTCanvas だけを追加したとすると、Frame のレイアウト機能は BorderLayout なので、QTCanvas だけがウインドウ一杯に表示されるはずです。そこで、ウインドウに対して pack メソッドを呼び出すことで、ウインドウのサイズを、QTCanvas、つまりはムービーのサイズに設定することができます。単なるビューアならこれで十分かもしれません。たとえば、ムービーを表示する MovieViewer というウインドウのもっとも簡単な定義はこうなります。

```
import java.awt.*;
import java.io.*;

import quicktime.app.*;
import quicktime.app.display.*;

public class MovieViewer   extends Frame
{
    private QTCanvas      qtBase;     //ムービーを配置するキャンバス
    private Drawable      qtCont;     //ファイルのムービーの内容

    public MovieViewer(File targetFile)
    {
        try   {
            qtBase = new QTCanvas();  //QuickTime の Canvas を用意する
            add(qtBase);              //Canvas をウインドウに追加する

            qtCont = QTFactory.makeDrawable(targetFile);
                                      //ムービーファイルから描画オブジェクトを取得する
```

```
//                      System.out.println(qtCont.getClass().getName());
                qtBase.setClient (qtCont, true);     //Canvas のクライアントにムービーを設定する
        }
        catch(Exception e)     {
                System.out.println(e.getMessage());
        }
        pack();     //ムービーの大きさに合わせてウインドウのサイズを調整する
        show();     //ウインドウを表示する
}
```

描画可能なオブジェクトの取得方法については、後で説明します。なお、別の部分で、QTSession.open をすでに呼び出して初期化はしてあります。流れとしては、QTCanvas を生成し、そのクライアントに描画可能なオブジェクトを設定するというだけです。ここで、BorderLayout の機能をそのまま使うので、pack により、ウインドウはムービーのサイズに最終的にはリサイズされます。同等なプログラムで生成されたウインドウは次のようになります。

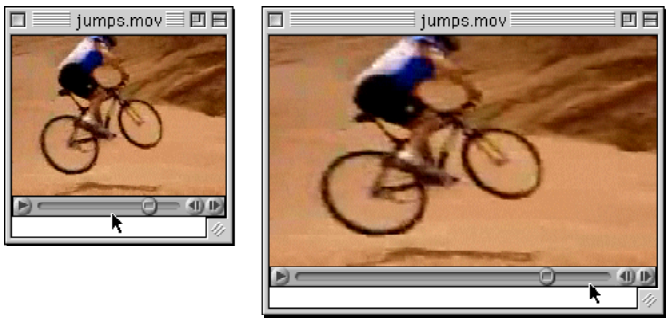

図 11.2　初期サイズ（左）と、ウインドウをリサイズした結果（右）

このように、pack メソッドで、初期状態ではムービーのサイズにウインドウの大きさは設定され、コントローラーもきちんと表示されています。コントローラーをクリックしてムービーをプレイしたり、コマを進めたり戻したりができるようになっています。

♦リサイズとアラインメント

QTCanvas を引数なしで生成すると、中に表示されるムービーは、QTCanvas の大きさに伸縮されて表示されます。一方、QTCanvas ではそうした表示だけでなく、一定の基準に従って、設定されるサイズを制限することができます。この設定を「リサイズのフラグ」と呼ぶことにします。リサイズのフラグとしては、次のようなクラス変数が QTCanvas クラスに用意されています。これらのクラス変数は、int 型です。複数の設定を行うために、クラス変数を加算することも可能です。引数なしで QTCanvas

を生成すると、リサイズのフラグは kFreeResize に設定されるということです。

表 11.3　リサイズのフラグとして設定可能なクラス変数

クラス変数	意味
QTCanvas.kIntegralResize	元のサイズの整数倍にしか縦や横のサイズは設定されない
QTCanvas.kInitialSize	元のサイズよりも小さくはなるが、大きくはならない
QTCanvas.kFreeResize	サイズは最小値よりも大きいサイズであれば任意に設定できる
QTCanvas.kAspectResize	縦横比は元データのまま変わらない
QTCanvas.kPerformanceResize	描画性能の高い状態、すなわち、サイズが 2 のべき乗サイズにしか設定できない
QTCanvas.kHorizontalResize	横幅を任意に設定できるが、高さは元データのサイズよりも小さくしかできない。VR ムービー向けの設定
QTCanvas.kVerticalResize	高さは任意に設定できるが、横は場は元データのサイズより小さくしかできない

リサイズフラグの中では、たとえば、kIntegralResize のように、元サイズの整数倍にしかサイズは設定されないようなものがあります。QTCanvas が整数倍の大きさでないような場合には、余白ができます。こうした余白ができる場合、クライアントの描画可能オブジェクトを QTCanvas の中でどう配置するかをきめるのが「アラインメント」という設定です。アラインメントは、縦方向、横方向、それぞれ 0 から 1.0 の間の float 型の値で指定することができます。アラインメントが縦横とも 0 なら、QTCanvas の左上の位置にぴったりとくっつくということです。

リサイズやアラインメントを設定するには、以下のような QTCanvas の機能を使います。QTCanvas のコンストラクタには、リサイズやアラインメントを作成できるものもあり、プログラムの途中で変更しないのなら、それを利用するのがいちばん手軽です。

表 11.4　QTCanvas のリサイズ設定とアラインメント関連の機能

戻り値	メソッド	機能
（コンストラクタ）	QTCanvas(int, float, float)	QTCanvas を生成するが、1 つ目の引数でリサイズに関するフラグを設定する（フラグは別表）。2 つ目の引数は横方向のアラインメント、3 つ目は縦方向のアラインメント
void	setResizeFlag(int)	リサイズの動作を引数で指定したフラグで指定する
int	getResizeFlag()	リサイズフラグの設定を得る

戻り値	メソッド	機能
void	setAlignment(float, float)	1 つ目の引数は横方向のアラインメント、2 つ目は縦方向のアラインメントを指定し、その値を設定する
void	setAlignmentX(float)	横方向のアラインメントを引数に指定した数値にする
void	setAlignmentY(float)	縦方向のアラインメントを引数に指定した数値にする
float	getAlignmentX()	横方向のアラインメントの値を取得する
float	getAlignmentY()	縦方向のアラインメントの値を取得する

たとえば、次のようなコンストラクタで、QTCanvas を作成したとします。

```
qtBase = new QTCanvas(QTCanvas.kIntegralResize, (float)0.3, (float)0.6);
                    //QuickTime の Canvas を用意する
```

つまり、ムービーのサイズは整数倍にしかならないわけです。こうして作成したウインドウは次のようになります。Frame の既定のレイアウト機能である BorderLayout を使い、pack メソッドをかけているので、初期状態は、ウインドウの大きさはムービーのサイズに設定されます。

図 11.3　整数倍にしかリサイズされないフラグを設定した QTCanvas

ここで、アラインメントは、横方向は 0.3、縦方向は 0.6 と指定しています。結果から分かるように、ムービーの中心の位置が、横方向は QTCanvas の幅の 0.3 の位置、

縦方向は QTCanvas の高さの 0.6 の位置に配置されます。アラインメントの値を利用して、このように、クライアントの QTCanvas 内での配置をアラインメントと言う属性で設定できるというわけです。

描画オブジェクトの生成

QTCanvas のクライアントになりうる描画可能なオブジェクトには、いくつかのクラスが定義されています。素直な方法では、そうしたクラスのオブジェクトを生成して…ということになるのですが、単にファイルやインターネットにあるムービーや画像ファイルから描画可能なオブジェクトを得るには、quicktime.app というパッケージにある QTFactory というクラスで定義されている以下のクラスメソッドを用いるのがいちばんの早道です。

表 11.5　QTFactory の描画オブジェクト生成クラスメソッド

戻り値	メソッド	機能
Drawable	QTFactory.makeDrawable(QTFile)	引数に指定したファイルから、描画オブジェクトを生成する
Drawable	QTFactory.makeDrawable(String)	引数に指定した URL から描画オブジェクトを生成する
Drawable	QTFactory.makeDrawable(InputStream, int, String)	引数に指定した InputStream からデータを取り出し、描画オブジェクトにする。2 つ目と 3 つ目の引数でデータの種類のヒントを指定するが、詳細は API のドキュメントを参照

makeDrawable は表以外にもメソッドの記述が可能ですが、それらは、描画可能なオブジェクトを生成するクラスを独自に定義した時に、そのクラスを指定することができるようになっているものです。

makeDrawable メソッドによって、Drawable オブジェクトが戻されるわけではありません。ムービーファイルを指定すると、quicktime.app.players.QTPlayer というオブジェクトが得られます。また、GIF ファイルなど画像ファイルを指定すると、quicktime.app.image.GraphicsImporterDrawer が得られます。これらは、Drawable を継承した QTDrawable をインプリメントしたクラスです。QTPlayer というのが文字通り、ムービーのプレイヤーで、ムービーやコントロールをひとまとめにしたオブジェクトと考えて良いでしょう。また、GraphicsImporterDrawer は画像ファイルのビューアとなる描画可能なオブジェクトです。

これらの特定用途の描画可能なオブジェクトをコンストラクタで生成するのもいいのですが、QTFactory のクラスメソッドを使うと、ファイルの種類を判断して、自動的にそれに合った描画可能なオブジェクトを生成して戻します。QTCanvas のクライアントを設定するメソッドは、Drawable なものなら何でも受け付けるので、こうしたQTFactroy を通じた方法が、ともかくいちばん簡単になるというわけです。

♦ファイルの扱い

makeDrawable メソッドでファイルを指定するには、java.io.File ではなく、quicktime.io パッケージにある QTFile クラスを指定する必要があります。このクラスのコンストラクタは File オブジェクトを引数に取るので、単に手続きの問題として QTFile オブジェクトを生成しておけば、makeDrawable メソッドが利用できます。それ以外に、QuickTime の標準ダイアログボックスを表示するクラスメソッドがあります。

表 11.6 QTFile のコンストラクタと標準ダイアログ

戻り値	メソッド	機能
（コンストラクタ）	QTFile(File)	引数に指定した File オブジェクトに対応したファイルのオブジェクトを生成する
（コンストラクタ）	QTFile(String)	指定したフルパス文字列に対応したファイルのオブジェクトを生成する
QTFile	QTFile.standardGetFilePreview (int[])	QuickTime が提供するファイルを開くための標準ダイアログボックスを表示する。引数に、一覧に表示するファイルのタイプを配列で指定するが、最大 4 つまで。要素が 0 の配列を指定するとすべてのファイルを一覧に表示する

standardGetFilePreview で標準ダイアログボックスを表示する時、ファイルタイプを int 型で指定します。ムービーファイルはファイルタイプが MooV ですが、これをいちいち int 型に自前で計算するのは面倒です。その場合は、MRJToolkit の機能を利用すると良いでしょう。たとえば、次のようにすると、ムービーファイルとテキストファイルだけに制限できます。

```
import com.apple.mrj.*;
import quicktime.io.*;

int acTypes[] = new int[2];
acTypes[0] = (new MRJOSType("MooV")).toInt();
```

```
acTypes[1] = (new MRJOSType("TEXT")).toInt();
QTFile    selectedFile = null;
try{
    selectedFile = QTFile.standardGetFilePreview(acTypes);
}
catch(Exception e)   {
    System.out.println(e.getMessage());
}
```

ただし、GIF ファイルや JPEG ファイルは、無条件に標準ダイアログボックスに表示されるようです。

もし、すべてのファイルを表示したいのあでれば、「int acType = new int[0];」のようにして要素が 0 個の配列を作成して、それを standardGetFilePreview の引数に指定すればいいでしょう。

11-3 ムービーをコントロールする

QTCanvas に描画可能なオブジェクトを独自に構成すれば、ムービーのプレイやストップといったコントロールや、コピー&ペーストなどの基本的な編集処理ができるようになります。いきなり描画可能なオブジェクトを生成するのではなく、一歩一歩作る方法として説明をしましょう。

ムービーデータの取得

ムービーそのものを扱う Movie クラスが、quicktime.std.movies というパッケージで定義されています。この Movie クラスを生成するにはコンストラクタも用意されていますが、生成のためのクラスメソッドがいくつか用意されています。ファイルや URL の Movie オブジェクトを生成するには、以下のメソッドが便利でしょう。

表 11.7　Movie を生成するクラスメソッド

戻り値	メソッド	機能
Movie	Movie.fromDataRef(DataRef, int)	1 つ目の引数に指定した DataRef に含まれるムービーの Movie オブジェクトを生成する。2 つ目の引数は動作フラグを設定するが、Toolbox 関数の NewMovieFromDataRef のフラグ定数を利用する

ここで DataRef というクラスが登場しますが、これは quicktime.std.movies.media というパッケージに定義されています。文字通り、ムービーのデータのもとになるものを参照するオブジェクトです。DataRef のコンストラクタとしては以下のようにいくつかのバリエーションがあります。ファイルや URL の文字列から作成できるので、これを使えば一般には問題ないでしょう。

表 11.8　DataRef クラスのコンストラクタ

コンストラクタ	引数の指定
DataRef(QTFile)	引数に指定したファイルを参照するオブジェクトを生成する
DataRef(String)	引数に指定した URL を参照するオブジェクトを生成する
DataRef(AliasHandle)	引数にはエイリアスを設定する（説明は省略）
DataRef(QTHandleRef)	引数にハンドルデータを指定する（説明は省略）
DataRef(QTHandleRef, int,	引数にハンドルデータを指定する（説明は省略）

コンストラクタ	引数の指定
String)	

たとえば、特定の URL から Movie オブジェクトを生成するには、次のようなプログラムになります。

```
import quicktime.std.movies.*;
import quicktime.std.movies.media.*;
    :
//変数 openURL が、java.net.URL クラスの変数で、開く URL が設定されているとする
DataRef  movieURL = new DataRef(openURL.toString());
                                        //ムービーデータを含む対象を参照する
Movie mv = Movie.fromDataRef(movieURL,StdQTConstants.newMovieActive);
                                        //ムービーデータを参照する
```

コントローラとプレイヤーの生成

QTCanvas に描画可能なオブジェクトには、コントローラが付きますが、構造の上では、ムービーにコントローラをかぶせて、それを描画可能オブジェクトとにするという形態を取ることができます。まず、コントローラーは、quicktime.std.movies というパッケージにある MovieController というクラスで管理できます。MovieController はコンストラクタを利用して生成します。通常は、Movie オブジェクトが生成されていれば、引数を 1 つだけ取るコンストラクタを利用することになるでしょう。

表 11.9　MovieController **のコントラクタ**

戻り値	メソッド	利用方法
コンストラクタ	MovieController(Movie)	引数に指定したムービーを管理するコントローラーを生成する
	MovieController(Movie, int)	1 つ目の引数に指定したムービーを管理するコントローラーを生成する。2 つ目の引数はフラグ
	MovieController(int, Movie, QDGraphics, QDPoint)	サブタイプを指定し、ウインドウを指定して、ムービーをコントローラとともに配置する（詳細は API のドキュメントを参照）
Movie	getMovie()	コントローラーの管理下にあるムービーを取得する

♦ムービーのプレイヤー

ムービーを実際に表示するのは、quicktime.app.players パッケージにある QTPlayer

というクラスです。文字どおりこのクラスがムービーのプレイヤーになります。このクラスは、Drawable クラスを継承しているので、描画可能なオブジェクトであり、QTCanvas にクライアントとして設定することができます。QTPlayer は、コンストラクタで生成しますが、以下にあるように、引数には MovieController を取ります。

表 11.10　QTPlayer **のコンストラクタ**

戻り値	メソッド	利用方法
コンストラクタ	QTPlayer(MovieController)	引数に指定したコントローラを持つムービーから、QTCanvas にクライアントとして設定可能なクラスを生成する
MovieController	getMovieController()	プレイヤーに含まれるコントローラを取得する

こうして生成した QTPlayer を QTCanvas に setClient メソッドを使って追加すれば、ウインドウなどに表示することができます。

なお、getMovieController は、QTFactory のクラスメソッドを使って描画可能なオブジェクトを得た場合、そのコントローラへの参照が必要になった時に利用することになるでしょう。次に説明するように、ムービーへの各種の作業は、コントローラーに対して行うので、いきなり描画可能オブジェクトを生成した場合には、こうしたメソッドを利用する必要が出てきます。

QTPlayer を使ってムービーを表示する場合のプログラムの流れは、たとえば次のリストのようになります。Frame クラスを継承したクラス内にあるプログラムで、変数 openURL に開くアドレスが URL 型で代入されているとします。全体的なプログラムは、後の 11.4 節のサンプルプログラムを参照してください。

```
private QTCanvas    qtBase;     //ムービーを配置するキャンバス
private QTDrawable moviePlay; //URL にあるムービー
private Movie          mv;
private DataRef       movieURL;
private MovieController movieCont;

    :

try {
    qtBase = new QTCanvas();  //QuickTime の Canvas を用意する
    add(qtBase);                    //Canvas をウインドウに追加する

    movieURL = new DataRef(openURL.toString());     //ムービーデータを含む対象を参照する
    mv = Movie.fromDataRef(movieURL,StdQTConstants.newMovieActive);
                                                    //ムービーデータを参照する
```

```
    movieCont = new MovieController(mv,0);      //ムービーを操作するコントローラを生成する
    movieCont.enableEditing(true);              //コントローラ上で編集可能にする
    movieCont.setKeysEnabled(true);             //キー操作の受付を可にする
    moviePlay = new QTPlayer(movieCont);
                                        //コントローラーから描画可能なオブジェクトを生成する
    qtBase.setClient(moviePlay, true); //Canvas のクライアントにムービーを設定する
}
catch(Exception e)  {
    System.out.println(e.getMessage());
}
```

その他のプレイヤー

コントロールを表示してムービーを再生するには、QTPlayer を使いましたが、コントロールを表示しないでムービーを表示し再生するプレイヤーのクラスも用意されています。quicktime.app.players パッケージに、MoviePlayer と MoviePresentor というクラスがあり、いずれも、ムービーをコントロールなしで表示できます。いずれもコンストラクタは Movie オブジェクトを引数に取り、指定した Movie を表示するプレイヤーとなります。いずれも、Drawable を継承しているので、QTCanvas のクライアントとして設定できます。

いずれのクラスも、単にムービーを表示するだけなので、たとえば表示したと同時に再生するようなことは、プログラムで記述する必要があります。いずれのクラスも、getMovie メソッドで、管理しているムービーを取得できるので、後は Movie クラスのメソッドを使って、ムービーの処理を行います。たとえば、再生するのなら、start メソッドを利用します。

表 11.11　MoviePlayer と MoviePresentor のコンストラクタと機能

クラス	戻り値	メソッド	機能
MoviePlayer	（コンストラクタ）	MoviePlayer(Movie)	引数のムービーを表示するための MoviePlayer オブジェクトを生成する
	Movie	getMovie()	中に含まれるムービーを取得する
MoviePresenter	（コンストラクタ）	MoviePresenter(Movie)	引数のムービーを表示するための MoviePresentor オブジェクトを生成する
	Movie	getMovie()	中に含まれるムービーを取得する

MoviePlayer を使ってムービーを表示するには、たとえば次のようなプログラムに

なります。Frame クラスを継承したクラス内にあるプログラムで、変数 openURL に開くアドレスが URL 型で代入されているとします。全体的なプログラムは、後の 11.4 節のサンプルプログラムを参照してください。

```
private QTCanvas    qtBase;     //ムービーを配置するキャンバス
private Movie       mv;
private DataRef     movieURL;
private MoviePlayer  moviePlayMP;

     :
try  {
     qtBase = new QTCanvas();  //QuickTime の Canvas を用意する
     add(qtBase);              //Canvas をウインドウに追加する

     movieURL = new DataRef(openURL.toString());     //ムービーデータを含む対象を参照する
     mv = Movie.fromDataRef(movieURL,StdQTConstants.newMovieActive);
                                                     //ムービーデータを参照する

     moviePlayMP = new MoviePlayer (mv); //MoviePlayer を生成する
     qtBase.setClient(moviePlayMP, true);    //Canvas のクライアントにムービーを設定する
     moviePlayMP.getMovie().start();         //自動的にスタートする
}
catch(Exception e)   {
     System.out.println(e.getMessage());
}
```

MoviePresenter も MoviePlayer と同様、単にムービーを表示するだけで、コントローラーは表示しません。このクラスを利用するとオフスクリーンのバッファを利用して描画するので、条件によっては高いパフォーマンスも得られるというものです。以下は、MoviePresenter を使ってウインドウにムービーを表示する例です。Frame クラスを継承したクラス内にあるプログラムで、変数 openURL に開くアドレスが URL 型で代入されているとします。全体的なプログラムは、後の 11.4 節のサンプルプログラムを参照してください。

```
private QTCanvas    qtBase;     //ムービーを配置するキャンバス
private Movie       mv;
private DataRef     movieURL;
private MoviePresenter     moviePresen;

     :
try  {
     qtBase = new QTCanvas();  //QuickTime の Canvas を用意する
     add(qtBase);              //Canvas をウインドウに追加する

     movieURL = new DataRef(openURL.toString());     //ムービーデータを含む対象を参照する
     mv = Movie.fromDataRef(movieURL,StdQTConstants.newMovieActive);
                                                     //ムービーデータを参照する
```

```
        moviePresen = new MoviePresenter (mv);
        qtBase.setClient(moviePresen, true);     //Canvas のクライアントにムービーを設定する
        moviePresen.getMovie().start();          //自動的にスタートする
    }
    catch(Exception e)  {
        System.out.println(e.getMessage());
    }
```

コントローラを通じてムービーをコントロール

ムービーを再生するなどのコントローラを使う機能は、MovieController クラスへのメソッドでも実現できます。次のようなメソッドがあります。プログラムでは、MovieController クラスの変数を、クラスのメンバ変数として用意しておき、いろいろなメソッドで利用できるようにしておくと便利でしょう。

表 11.12　MovieContorlller **クラスで使える機能**

戻り値	メソッド	機能
void	play(folat)	ムービーの再生を行う。引数で方向や再生速度の割合を指定する。1 なら通常通り、0.5 なら半分の速さ、-1 なら逆行
void	step(int)	現在の再生位置のポインタを、引数に指定したフレーム数だけ前後に移動する。引数が 0 なら、1 フレーム前に進める
void	stop()	ムービーの再生を止める
void	prerollAndPlay(float)	最初に戻して再生を行う。引数は、play メソッドと同様（このメソッドは機能していない？）
void	setPlaySelection(boolean)	選択範囲を再生するか、全体を再生するかを引数で指定する。true なら選択範囲を再生する
boolean	getPlaySelection()	再生を選択範囲に対して行うなら true、全体に対して行うなら false が得られる
int	getCurrentTime()	現在のポインタの位置を取得する
void	goToTime(TimeRecord)	現在のポインタを指定した時間位置に設定する
void	setVolume(float newVolume)	再生音量を引数の値に設定する。引数は-1～1の値を取るが、負の数の時には音は出さない。現在の音量をキープしたまま音をとめる時に負の数を利用する
float	getVolue()	現在の再生音量を取得する
void	setLooping(boolean)	ループ再生するかどうかを引数で指定する
boolean	getLooping()	ループ再生するかどうかを取得する

♦ムービーの編集作業

QuickTime に付属する Movie Player やあるいは QuickTime Player では、ムービーの

コピー&ペーストなど、さまざまに編集作業ができます。基本的な編集作業については、次の表に示すような MovieController のメソッドを使って行うことができます。ここで、カットやコピーのメソッドが Movie 型の戻り値を持っていることに注意をしてください。つまり、cut や copy メソッドでは、システムのスクラップにはデータは入らないのです。他のアプリケーションにムービーデータを持って行くのであれば複雑になりますが、同一のアプリケーション内だけでコピー&ペーストを実現するのなら、プライベートのスクラップとして、適当に Movie 型変数を設けておき、cut や copy の戻り値はその変数に代入しておきます。そして、ペーストの時、プライベートスクラップがある場合には、その値をペーストしますが、ない場合にはシステムスクラップの内容をペーストします。ただし、厳密には、アクティベートやディアクティベートのタイミングなどから、別のアプリケーションによってシステムのスクラップが変更されていないのかなどを取得する必要があります。

表 11.13　MovieController **のムービーの編集に関する機能**

戻り値	メソッド	機能
void	enableEditing(boolean enabled)	ムービーの編集処理を可能にするかどうかを引数で指定する
boolean	isEditingEnabled()	ムービーが編集可能かどうかを取得する
Movie	getMovie()	コントローラの管理下にあるムービーを取得する
void	setSelectionBegin(TimeRecord)	選択範囲の先頭位置を設定する
TimeRecord	getSelectionBegin()	選択範囲の先頭位置を取得する
void	setSelectionDuration(TimeRecord)	選択範囲の長さを設定する
TimeRecord	getSelectionDuration()	選択範囲の長さを設定する
Movie	copy()	選択範囲のムービーを取得し、戻り値として戻す
Movie	cut()	選択範囲のムービーを取得し、戻り値として戻す。さらに選択範囲は削除する
void	paste()	システムのスクラップの内容を現在の選択範囲に貼り付ける
void	paste(Movie)	引数に指定したムービーを、現在の選択範囲に貼り付ける
void	clear()	現在の選択範囲を消去する
void	undo()	コントローラに対して行った編集処理に対して行った前の状態に戻す

ムービー自体の処理

Movie クラスには、コントローラー以上の様々な編集処理ができるようになってい

ます。利用しそうなメソッドをピックアップしておきましょう。

表 11.14　Movie クラスのムービー再生などに関する機能

戻り値	メソッド	機能
void	start()	ムービーを再生する
void	stop()	ムービーの再生を停止する
void	goToBeginning()	ポインタを先頭に移動する
void	goToEnd()	ポインタを最後に移動する
int	getTime()	ポインタの位置をタイムスケールで得る
void	setTIme(TimeRecord)	ポインタの位置を引数で設定する。TimeRecord についてはドキュメントを参照
boolean	isDone()	すべてを再生し終えたかどうかを戻す
void	preroll(int, float)	ポインタを 1 つ目の引数で指定した位置に戻し、2 つ目の引数を再生時の割合（MovieController の play メソッドの引数と同じ意味）に設定する

表 11.15　Movie クラスのムービーの情報に関する機能

戻り値	メソッド	機能
QDRect	getBounds()	ムービーの表示領域を取得する。QDRect については java.awt.Rect を拡張したもので、詳細はドキュメントを参照
void	setBounds(QDRect)	ムービーの表示領域を設定する
int	getTimeScale()	タイムスケールを取得する（タイムスケールは、ムービーの時間測定の基本単位で、1 秒を何分割するかという値）
void	setTimeScale(int)	タイムスケールを引数の値に設定する
int	getDuration()	タイムスケールでのムービーの長さを取得する
float	getRate()	再生レートを取得する
void	setRate(float)	再生レートを設定する
float	getVolume()	現在のボリュームの値を取得する
void	setVolume(float)	ボリュームを引数の値に設定する

表 11.16　Movie クラスの選択範囲と編集に関する機能

戻り値	メソッド	機能
TimeInfo	getSelection()	現在の選択範囲を取得する。TimeInfo は、時刻と長さを持つクラス、詳細はドキュメントを参照
void	setSelection(TimeInfo)	選択範囲を引数で指定した範囲に設定する
void	setSelection(int, int)	選択範囲を引数で指定した範囲に設定する。1 つ目の引数が選択範囲の最初の位置で、2 つ目が長さ。単位

		はタイムスケール
Movie	cutSelection()	現在の選択範囲のムービーを取得し、ムービーへの参照を戻す。選択範囲は削除される
Movie	copySelection()	現在の選択範囲のムービーを取得し、ムービーへの参照を戻す
void	pasteSelection(Movie)	引数に指定したムービーを、現在の選択範囲に上書きで貼り付ける
void	addSelection(Movie)	引数に指定したムービーを、現在の選択範囲に追加する
void	clearSelection()	現在の選択範囲を消去する

表 11.17　Movie **クラスのファイル化に関連する機能**

戻り値	メソッド	機能
Movie	flattenData(int, QTFile, int, int, int)	フラット化したムービーを作成する。引数は順にフラット化のフラグ、作成するファイル、クリエイター、スクリプトタグ、ファイル作成フラグを指定する（詳細はドキュメントを参照）。作成したムービーファイルのムービーを戻す
int	convertToFile(QTFile, int, int, int)	ムービーをファイルに保存する。引数は順に、作成するファイル、ファイルタイプ、クリエイタ、スクリプトタグ（詳細はドキュメントを参照）。作成したのリソース ID を戻す
int	convertToFile(QTFile, int, int, int, int)	ムービーをファイルに保存する。引数は順に、作成するファイル、ファイルタイプ、クリエイタ、スクリプトタグ、フラグ（詳細はドキュメントを参照）。作成したのリソース ID を戻す

11-4 サンプルプログラム

ムービーのビューアをサンプルプログラムとして作成しました。QuickTime for Java **の機能を使って** QTCanvas **に表示させるので、ムービーだけでなく画像ファイルも表示できますし、テキストもムービーとして表示させることができます。**

MovieViewer.java

ビューアの本体になるクラスで、Frame を継承して、つまりはムービーを表示可能なウインドウを定義したというわけです。ムービーをファイルや URL から表示し、コピー&ペーストなどの処理を行っています。なお、ファイルからと URL を指定しての処理形態はコンストラクタで切り替えており、処理内容は違っています。ファイルからは 11-2 で説明した方法、URL からは 11-3 で説明した方法を使っていますが、いずれの手法もファイルや URL からのムービー表示は可能です。

ムービーを表示すると、QTPlayer を使っている場合にはコントロールが出てくるので、手作業で再生や停止ができます。また、「編集」メニューの「コントロール」から、再生や逆行などができるようになっています。また、ムービーの編集を可能にしてあるので、ムービーのコピー&ペーストができるようにもなっています。

メニュー定義などが含まれているので長いプログラムになってはいますが、QuickTime for Java 自体の処理はそれほど組み込まれているわけではなく、たとえば、「コピー」だと 1 つのメソッドを呼び出すだけです。

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;
import java.util.*;

import quicktime.*;
import quicktime.io.*;

import quicktime.app.*;
import quicktime.app.display.*;
import quicktime.app.players.*;

import quicktime.std.*;
import quicktime.std.movies.*;
```

```
import quicktime.std.movies.media.*;

public class MovieViewer   extends Frame
    implements WindowListener
{
    private File            openFile = null;  //開いているファイル
    private URL             openURL = null; //開いているアドレス
    private Dimension       initilaWindowSize = new Dimension(500,300);
                                                        //ウインドウの初期サイズ

    private QTCanvas        qtBase;     //ムービーを配置するキャンバス

    private Drawable        qtCont;     //ファイルのムービーの内容

    private QTPlayer moviePlay;         //URL にあるムービー
    private Movie           mv;
    private DataRef  movieURL;
    private MovieController movieCont;
    private MoviePlayer     moviePlayMP;
    private MoviePresenter      moviePresen;

    private Movie           clipboardMovie = null; //クリップボードのムービー

    public MovieViewer()  {
        addWindowListener(this);    //WindowListener を組み込む
        setupMenuBar();             //メニューを構築する
        show();                     //ウインドウを表示する
        setSize(initilaWindowSize); //ウインドウのサイズを初期サイズにする
    }

    public MovieViewer(QTFile targetFile)
    {
        this();
        openFile = targetFile;  //引数のファイルをメンバ変数に保存

        /*ウインドウの表示など、ウインドウの初期化作業*/
        if (openFile != null)     {     //ファイルが指定されているかをチェック
            setTitle(openFile.getName());       //ウインドウのタイトルはファイル名にする
        }

        try   {
            qtBase = new QTCanvas();  //QuickTime の Canvas を用意する
            add(qtBase);                    //Canvas をウインドウに追加する

            qtCont = QTFactory.makeDrawable(targetFile);
                                    //ムービーファイルから描画オブジェクトを取得する
                System.out.println(qtCont.getClass().getName());
            qtBase.setClient (qtCont, true);    //Canvas のクライアントにムービーを設定する

            movieCont = ((QTPlayer)qtCont).getMovieController();
                                                    //コントローラを参照しておく
            movieCont.enableEditing(true);          //コントローラ上で編集可能にする
        }
        catch(Exception e)     {
```

```
            System.out.println(e.getMessage());
        }
        pack();     //ムービーの大きさに合わせてウインドウのサイズを調整する
    }

    public MovieViewer(java.net.URL targetURL)
    {
        this();
        openURL = targetURL;        //引数のアドレスをメンバ変数に保存

        if (openURL != null)    {       //アドレスが指定されているかをチェック
            setTitle(openURL.toString());       //ウインドウのタイトルはアドレスにする
        }

        try    {
            qtBase = new QTCanvas(QTCanvas.kIntegralResize, (float)0.3,(float)0.6);
                                        //QuickTime の Canvas を用意する
            add(qtBase);                //Canvas をウインドウに追加する

            movieURL = new DataRef(openURL.toString());
                                        //ムービーデータを含む対象を参照する
            mv = Movie.fromDataRef(movieURL,StdQTConstants.newMovieActive);
                                        //ムービーデータを参照する

        /*  以下、[1] [2] [3]のいずれかのブロック以外はコメントとしておき、
            実行されないようにする     */

            /*      [1] QTPlayer を使った描画   */
            movieCont = new MovieController(mv,0);
                                            //ムービーを操作するコントローラを生成する
            movieCont.enableEditing(true);          //コントローラ上で編集可能にする
            movieCont.setKeysEnabled(true);         //キー操作の受付を可にする
            moviePlay = new QTPlayer(movieCont);
                                    //コントローラーから描画可能なオブジェクトを生成する
            qtBase.setClient(moviePlay, true);
                                    //Canvas のクライアントにムービーを設定する

            /*      [2] MoviePlayer を使った描画       */
/*          moviePlayMP = new MoviePlayer (mv);
            qtBase.setClient(moviePlayMP, true);
                                    //Canvas のクライアントにムービーを設定する
            moviePlayMP.getMovie().start();         //自動的にスタートする
*/
            /*      [3] MoviePresenter を使った描画   */
/*          moviePresen = new MoviePresenter (mv);
            qtBase.setClient(moviePresen, true);
                                    //Canvas のクライアントにムービーを設定する
            moviePresen.getMovie().start();         //自動的にスタートする
*/      }
        catch(Exception e)      {
            System.out.println(e.getMessage());
        }
        pack();     //ムービーの大きさに合わせてウインドウのサイズを調整する
    }
```

```
private void setupMenuBar()
{
        MenuItem newItem, openItem, openURLItem, saveItem, saveAsItem,
                        closeItem, quitItem;     //「ファイル」メニューの項目
        MenuItem undoItem, cutItem, copyItem, pasteItem; //「編集」メニューの項目
        MenuItem cPlayNormal, cPlayBackword, cStop, cPreRoll;
                                    //「編集」メニューの「コントロール」のサブメニュー項目
        MenuItem aboutMRJ;          //「ヘルプ」メニューに追加する項目

        MenuBar mb = new MenuBar();    //メニューバーを用意する

        /*「ファイル」メニューの作成*/
        Menu fileMenu = new Menu("ファイル", true);        //「ファイル」メニューを作成する
        newItem = new MenuItem("新規", new MenuShortcut('N'));
                                            //「新規」項目を作成、ショートカットは N
        fileMenu.add(newItem);              //「ファイル」メニューに追加する
        openItem = new MenuItem("開く...", new MenuShortcut('O'));
                                            //「開く」項目を作成、ショートカットは O
        fileMenu.add(openItem);             //「ファイル」メニューに追加する
        openURLItem = new MenuItem("アドレスを開く...");
                                        //「アドレスを開く」項目を作成、ショートカットはなし
        fileMenu.add(openURLItem);          //「ファイル」メニューに追加する
        closeItem = new MenuItem("閉じる", new MenuShortcut('W'));
                                            //「閉じる」項目を作成、ショートカットは W
        fileMenu.add(closeItem);            //「ファイル」メニューに追加する
        fileMenu.addSeparator();            //区切り線を追加する
        saveItem = new MenuItem("保存", new MenuShortcut('S'));
                                            //「保存」項目を作成、ショートカットは S
        saveItem.setEnabled(false); //「保存」を選択できないようにイネーブルでなくす
        fileMenu.add(saveItem);             //「ファイル」メニューに追加する
        saveAsItem = new MenuItem("名前を付けて保存...");
                                        //「名前を付けて保存」項目を作成、ショートカットはなし
        saveAsItem.setEnabled(false);       //選択できないようにイネーブルでなくす
        fileMenu.add(saveAsItem);           //「ファイル」メニューに追加する
        fileMenu.addSeparator();            //区切り線を追加する
        quitItem = new MenuItem("終了", new MenuShortcut('Q'));
                                            //「終了」項目を作成、ショートカットは Q
        fileMenu.add(quitItem);             //「ファイル」メニューに追加する
        mb.add(fileMenu);               //「ファイル」メニューをメニューバーに追加する

        NewItemListener newListener = new NewItemListener();
                        //「新規」を選択した時にイベントを受け付けるクラスを新たに生成
        newItem.addActionListener(newListener);
                        //「新規」を選んだ時にイベントが発生するように設定

        OpenItemListener openListener = new OpenItemListener();
                        //「開く」を選択した時にイベントを受け付けるクラスを新たに生成
        openItem.addActionListener(openListener);
                        //「開く」を選んだ時にイベントが発生するように設定

        OpenURLItemListener openURLListener = new OpenURLItemListener();
            //「アドレスを開く」を選択した時にイベントを受け付けるクラスを新たに生成
        openItem.addActionListener(openURLListener);
```

```
        //「アドレスを開く」を選んだ時にイベントが発生するように設定

CloseItemListener closeListener = new CloseItemListener();
        //「閉じる」を選択した時にイベントを受け付けるクラスを新たに生成
closeItem.addActionListener(closeListener);
        //「閉じる」を選んだ時にイベントが発生するように設定

SaveItemListener saveListener = new SaveItemListener();
        //「保存」「保存を付けて保存」を選択した時にイベントを受け付けるクラスを
        //      新たに生成
saveItem.addActionListener(saveListener);
        //「保存」を選んだ時にイベントが発生するように設定
saveAsItem.addActionListener(saveListener);
        //「保存を付けて保存」を選んだ時にイベントが発生するように設定

QuitItemListener quitListener = new QuitItemListener();
        //「終了」を選択した時にイベントを受け付けるクラスを新たに生成
quitItem.addActionListener(quitListener);
        //「終了」を選んだ時にイベントが発生するように設定

/*「編集」メニューの作成*/
Menu editMenu = new Menu("編集", true);      //「編集」メニューを作成する
undoItem = new MenuItem("やり直し", new MenuShortcut('Z'));
                                        //「やり直し」項目を作成、ショートカットは Z
editMenu.add(undoItem);             //「編集」メニューに追加する
editMenu.addSeparator();            //区切り線を追加する
cutItem = new MenuItem("カット", new MenuShortcut('X'));
                                        //「カット」項目を作成、ショートカットは X
editMenu.add(cutItem);              //「編集」メニューに追加する
copyItem = new MenuItem("コピー", new MenuShortcut('C'));
                                        //「コピー」項目を作成、ショートカットは C
editMenu.add(copyItem);             //「編集」メニューに追加する
pasteItem = new MenuItem("ペースト", new MenuShortcut('V'));
                                        //「ペースト」項目を作成、ショートカットは V
editMenu.add(pasteItem);            //「編集」メニューに追加する
editMenu.addSeparator();            //区切り線を追加する

/*「編集」メニュー「コントロール」でのサブメニューの作成*/
Menu bgControlMenu = new Menu("コントロール");
                                        //サブメニューの元になる「コントロール」項目を作成
bgControlMenu.add(cPlayNormal = new MenuItem("プレイ")); //「プレイ」項目を作成
bgControlMenu.add(cPlayBackword = new MenuItem("逆行")); //「逆行」項目を作成
bgControlMenu.add(cStop = new MenuItem("ストップ"));        //「ストップ」項目を作成
bgControlMenu.add(cPreRoll = new MenuItem("プリロール"));
                                            //「プリロール」項目を作成
editMenu.add(bgControlMenu);
                        //「コントロール」のサブメニューを「編集」メニューに追加する

mb.add(editMenu);       //「編集」メニューをメニューバーに追加する
setMenuBar(mb);         //メニューバーを Frame に設定する

EditItemListener editListener = new EditItemListener();
        //「編集」メニューを選択した時の処理を行うクラスを新たに生成する
undoItem.addActionListener(editListener);
```

```
        //「やり直し」を選択した時にイベントが発生するようにしておく
    cutItem.addActionListener(editListener);
        //「カット」を選択した時にイベントが発生するようにしておく
    copyItem.addActionListener(editListener);
        //「コピー」を選択した時にイベントが発生するようにしておく
    pasteItem.addActionListener(editListener);
        //「ペースト」を選択した時にイベントが発生するようにしておく

    ControlItemListener bgListener = new ControlItemListener();
        //「背景色」のサブメニューを選択した時の処理を行うクラスを新たに生成する
    cPlayNormal.addActionListener(bgListener);
        //「プレイ」を選択した時にイベントが発生するようにしておく
    cPlayBackword.addActionListener(bgListener);
        //「逆行」を選択した時にイベントが発生するようにしておく
    cStop.addActionListener(bgListener);
        //「ストップ」を選択した時にイベントが発生するようにしておく
    cPreRoll.addActionListener(bgListener);
        //「プリロール」を選択した時にイベントが発生するようにしておく

    /*「ヘルプ」メニューへの追加*/
    Menu helpMenu = new Menu("ヘルプ"); //「ヘルプ」メニューを作成する
    helpMenu.add(aboutMRJ = new MenuItem("MJR とは？"));
                        //「MJR とは？」という項目を追加する
    mb.add(helpMenu);    //「ヘルプ」メニューをメニューバーに追加する
    mb.setHelpMenu(helpMenu);       //ここで作成したメニューをヘルプメニューとする
        //これにより、システムが用意する「ヘルプ」メニューと一体化する

    aboutMRJ.addActionListener(
            //ヘルプメニューに追加した項目を選択した時の処理を直接記述している
        new ActionListener()        {
            public void actionPerformed(ActionEvent ev)  {
                System.out.println("Help");   //単にコンソールに文字を出力するだけ
            }    }    );
}

/*「ファイル」メニューの「新規」を選択した時の処理*/
class NewItemListener implements ActionListener  {
    public void actionPerformed(ActionEvent ev)  {
        new MovieViewer();    //TextViewer のウインドウを新たに作成する
    }
}

/*「ファイル」メニューの「開く」を選択した時の処理*/
class OpenItemListener implements ActionListener {
    public void actionPerformed(ActionEvent ev)  {
        int acTypes[] = new int[0];
        QTFile selectedFile = null;
        try{
            selectedFile = QTFile.standardGetFilePreview(acTypes);
        }
        catch(Exception e)    {
            System.out.println(e.getMessage());
        }
        if(selectedFile != null)  //ファイルを選択したのなら
```

```
                    new MovieViewer(selectedFile);
        }
}

/*「ファイル」メニューの「アドレスを開く」を選択した時の処理*/
class OpenURLItemListener implements ActionListener   {
        public void actionPerformed(ActionEvent ev)  {
                InputURL inputDlg = new InputURL(MovieViewer.this);
                inputDlg.show();
                String urlString = inputDlg.getURL();
                if (urlString != null)
                        try    {
                                new MovieViewer(new URL(urlString));
                        }
                        catch(Exception e)     {
                                System.out.println(e.getMessage());
                        }
                inputDlg.dispose();
        }
}

/*「ファイル」メニューの「閉じる」を選択した時の処理*/
class CloseItemListener implements ActionListener {
        public void actionPerformed(ActionEvent ev)  {
                dispose();          //Frame を閉じる
        }
}

/*「ファイル」メニューの「保存」「名前を付けて保存」を選択した時の処理*/
class SaveItemListener implements ActionListener  {
        public void actionPerformed(ActionEvent ev)  {
                /*      未作成       */
        }
}

/*「ファイル」メニューの「終了」を選択した時の処理*/
class QuitItemListener implements ActionListener   {
        public void actionPerformed(ActionEvent ev)  {
                QTSession.close();     //QuickTime を終了する
                System.exit(0);
        }
}

/*「編集」メニューを選択した時の処理*/
class EditItemListener implements ActionListener   {

        public void actionPerformed(ActionEvent ev)  {
                String itemLabel = ((MenuItem)ev.getSource()).getLabel();
                                                        //選択したメニューの項目名を取り出す
                try    {
                        if(itemLabel.compareTo("やり直し") == 0)   //選択したのが「やり直し」なら
                                movieCont.undo();
                        else if(itemLabel.compareTo("カット") == 0)   //選択したのが「カット」なら
                                clipboardMovie = movieCont.cut();
```

```
                else if(itemLabel.compareTo("コピー") == 0)
                        clipboardMovie = movieCont.copy();
                else if(itemLabel.compareTo("ペースト") == 0)
                                                        //選択したのが「ペースト」なら
                        if(clipboardMovie == null)
                                movieCont.paste();
                        else
                                movieCont.paste(clipboardMovie);
                }
                catch(Exception e)      {
                        System.out.println(e.getMessage());
                }
        }
    }

    /*「編集」メニューの「背景色」から項目をを選択した時の処理*/
    class ControlItemListener implements ActionListener       {
        public void actionPerformed(ActionEvent ev)  {
                String itemLabel = ((MenuItem)ev.getSource()).getLabel();
                                                    //選択したメニューの項目名を取り出す
                try    {
                        if(itemLabel.compareTo("プレイ") == 0)          //選択したのが「プレイ」なら
                                movieCont.play((float)1);
                        else if(itemLabel.compareTo("逆行") == 0)      //選択したのが「逆行」なら
                                movieCont.play((float)-0.5);
                        else if(itemLabel.compareTo("ストップ") == 0)
                                                        //選択したのが「ストップ」なら
                                movieCont.play((float)-0);
                        else if(itemLabel.compareTo("プリロール") == 0)
                                                        //選択したのが「プリロール」なら
                                movieCont.prerollAndPlay((float)1);
                }
                catch(Exception e)      {
                        System.out.println(e.getMessage());
                }
        }
    }

    /*ウインドウのクローズボックスをクリックした時*/
    public void windowClosing(WindowEvent e)
    {
        dispose();          //ウインドウを閉じる。
                    //これがないとクローズボックスをクリックしてもウインドウは閉じない
    }
    public void windowOpened(WindowEvent e){}
    public void windowClosed(WindowEvent e){}
    public void windowIconified(WindowEvent e){}
    public void windowDeiconified(WindowEvent e){}
    public void windowActivated(WindowEvent e){}
    public void windowDeactivated(WindowEvent e){}
}
```

Draggable.java

ビューアプリケーションの main を含むクラスで、AppleEvent の処理などを組み込んでいます。第 5 章で説明した、Mac OS 向けのアプリケーションの土台となる部分です。QuickTime を初期化したり、ファイルを開くダイアログボックスあたりは QuickTime for Java に関わってくる部分ですが、多くのプログラムは、これまでの章で説明してきたことと大きく変わるところはありません。

```
import com.apple.mrj.*;
import quicktime.*;
import quicktime.io.*;
import java.io.*;
import java.net.*;
import java.util.*;
import java.awt.*;
import java.awt.event.*;

public class Draggable extends Frame
    implements      MRJOpenDocumentHandler,
                    MRJPrintDocumentHandler,
                    MRJQuitHandler,
                    MRJAboutHandler
{
    static public void main(String arg[])      //起動時に実行されるメソッド
    {
        try    {
            QTSession.open();     //QuickTime を初期化する
        }
        catch(Exception e)     {
            System.out.println(e.getMessage());
        }
        new Draggable();       //このクラスを新たに生成する
    }

    public Draggable()     //コンストラクタ
    {
        /*基本 AppleEvent が、このクラスに伝達されるように定義する*/
        MRJApplicationUtils.registerOpenDocumentHandler(this);
        MRJApplicationUtils.registerPrintDocumentHandler(this);
        MRJApplicationUtils.registerQuitHandler(this);
        MRJApplicationUtils.registerAboutHandler(this);

        setupMenuBar(); //メニューバーの設定

        setBounds(-1000,-100,10,10);     //ウインドウの位置を画面の外側に出す。つまり非表示
            //にしたいのだが、hide()とするとメニューまで消えるので、こういう方針にした
        show();     //ウインドウを表示するが、実際には画面には見えない
    }
```

```
    public void handleOpenFile(File filename)       //Open イベントが Finder からやってきた時
    {
        System.out.println("Dragged "+filename.toString());
        MovieViewer newViewer = new MovieViewer(new QTFile(filename));
                                                    //開くファイルを TextViewer で表示する
    }

    public void handlePrintFile(File filename)      //Print イベントが Finder からやってきた時
    {
        System.out.println("Print Event "+filename.toString());
        System.out.println(" Sorry, Not supporting...");
    }

    public void handleQuit()        //Quit イベントが Finder からやってきた時、
                                    //あるいはアップルメニューの「終了」を選択した時
    {
        System.out.println("Quit Event Received");
        QTSession.close();      //QuickTime を終了する
        System.exit(0);   //アプリケーションの終了
    }

    public void handleAbout()     //アップルメニューの About を選択した時
    {
        System.out.println("Select About Menu");
    }

/*TextViewer のウインドウが何も表示されていない時にもメニューを表示されるようにしたい。
そのような場合に必要なメニューに絞って表示するが、メニューを表示するにはそもそも Frame
でなければならない。このクラスは Frame を拡張したが、そのウインドウは表示したくないので、
コンストラクタで画面外に追いやった        */

    private void setupMenuBar()
    {
        /*メニュー作成部分のコメントは TextViewer.java を参照*/
        MenuItem newItem, openItem, openURLItem, closeItem, quitItem;

        MenuBar mb = new MenuBar();

        Menu fileMenu = new Menu("ファイル", true);
        newItem = new MenuItem("新規", new MenuShortcut('N'));
        fileMenu.add(newItem);
        openItem = new MenuItem("開く...", new MenuShortcut('O'));
        fileMenu.add(openItem);
        openURLItem = new MenuItem("アドレスを開く...");
        fileMenu.add(openURLItem);
        fileMenu.addSeparator();
        quitItem = new MenuItem("終了", new MenuShortcut('Q'));
        fileMenu.add(quitItem);
        mb.add(fileMenu);

        NewItemListener newListener = new NewItemListener();
        newItem.addActionListener(newListener);

        OpenItemListener openListener = new OpenItemListener();
```

```
        openItem.addActionListener(openListener);

        OpenURLItemListener openURLListener = new OpenURLItemListener();
        openURLItem.addActionListener(openURLListener);

        QuitItemListener quitListener = new QuitItemListener();
        quitItem.addActionListener(quitListener);

        setMenuBar(mb);
    }

    class NewItemListener implements ActionListener   {
        public void actionPerformed(ActionEvent ev)   {
            new MovieViewer();
        }
    }

    class OpenItemListener implements ActionListener {
        public void actionPerformed(ActionEvent ev)   {
            int acTypes[] = new int[2];
            acTypes[0] = (new MRJOSType("MooV")).toInt();
            acTypes[1] = (new MRJOSType("TEXT")).toInt();
            QTFile selectedFile = null;
            try{
                selectedFile = QTFile.standardGetFilePreview(acTypes);
            }
            catch(Exception e)      {
                System.out.println(e.getMessage());
            }
            if(selectedFile != null)   //ファイルを選択したのなら
                new MovieViewer(selectedFile);
        }
    }

    /*「ファイル」メニューの「アドレスを開く」を選択した時の処理*/
    class OpenURLItemListener implements ActionListener   {
        public void actionPerformed(ActionEvent ev)   {
            InputURL inputDlg = new InputURL(Draggable.this);
            inputDlg.show();
            String urlString = inputDlg.getURL();
            if (urlString != null)
                try     {
                    new MovieViewer(new URL(urlString));
                }
                catch(Exception e)      {
                    System.out.println(e.getMessage());
                }
            inputDlg.dispose();
        }
    }

    class QuitItemListener implements ActionListener   {
        public void actionPerformed(ActionEvent ev)   {
            System.exit(0);
```

```
        }
    }
}
```

InputURL.java

「ファイル」メニューから「アドレスを開く」を選択した時、開くアドレスを指定するダイアログボックスを定義したのがこのクラスです。単に、TextField や Button があるだけの単純なダイアログボックスです。

なお、ダイアログボックスのアドレスの初期設定は、http://msyk.locus.co.jp/mjr/VCL00001.AVI となっていますが、実際にこのアドレスにはムービーが存在します。極力、削除はしないようにするので、サンプルとして実行したい場合には、このアドレスのデータを御利用ください。

```
import java.awt.*;
import java.awt.event.*;

public class InputURL extends Dialog
{
    private TextField urlField;
    private Button okButton, cancelButton;

    public InputURL(Frame myParent)        {
        super(myParent, true);
        setSize(310,60);
        setLayout(null);
        setTitle("開く QuickTime ムービーのアドレスを指定してください：");

        urlField = new TextField("http://msyk.locus.co.jp/mjr/VCL00001.AVI");
        urlField.setBounds(5,5,300,20);
        add(urlField);

        okButton = new Button("開く");
        okButton.setBounds(150,30,70,25);
        add(okButton);
        okButton.addActionListener(new ActionListener()    {
            public void actionPerformed(ActionEvent ev)  {
                hide();
            }
        }            );

        cancelButton = new Button("キャンセル");
        cancelButton.setBounds(230,30,70,25);
        add(cancelButton);
        cancelButton.addActionListener(new ActionListener()      {
            public void actionPerformed(ActionEvent ev)  {
                hide();
                urlField.setText("");
```

```
                }
            }           );
    }

    public String getURL() {       //テキストフィールドに入力したアドレスを取り出す
        String fieldString = urlField.getText();
        if(fieldString.equals(""))
            return null;
        else
            return urlField.getText();
    }
}
```

（第 11 章以上）

LOCUS Report

第12章 ネイティブライブラリの呼び出し

Java のプログラムから、C や C++などで作ったライブラリ内の関数を呼び出す事が可能です。

Java の基本ライブラリに用意されている JNI の機能を使い、Mac OS 上に作成した共有ライブラリを呼び出す方法を説明します。

CodeWarrior Pro4 を使って、Java のアプリケーションと、呼び出される共有ライブラリの作成方法を具体的に説明しましょう。

12-1 Java から C/C++のプログラムを使う

ネイティブメソッドの呼び出しの意味と、非常に簡単なプログラムながら、CodeWarrior Pro4 でどのようにして Java 側のプログラムと共有ライブラリを作成するのかを説明します。CodeWarrior を使ってのプログラム作成の方法を具体的に説明をしましょう。

ネイティブな呼び出しとその必要性

Java の基本ライブラリには、AWT などを始め、かなりの機能が揃っており、一般的なアプリケーションの作成などではそれほど困ることはないレベルだと言えるでしょう。しかしながら、必要な機能がすべて揃っているわけではありません。ない機能はプログラムとして用意しなければなりませんが、そのすべてを Java で構築できるとは限りません。たとえば、Mac OS の API をコールしないといけないような場合だと、Java のプログラムからは OS の API を直接呼び出すことができません。また、ハードウエアを直接操作するようなプログラムも Java では作れないに等しい状態です。

こうした用途も踏まえて、Java では、C や C++などの言語で作成されたライブラリソフトウエアを呼び出す機能を JNI（Java Native Interface）として備えています。Java は Java VM 上で動作しているのに対し、一般には C/C++などで作られたプログラムは動作ターゲットの CPU 向けのバイナリコードになっています。こうした特定 CPU をターゲットにしたプログラムを「ネイティブコード」などと表現されます。C や C++で作った関数は「ネイティブ関数」、一連のネイティブコードを含むライブラリを「ネイティブライブラリ」と呼ぶことにします。

JNI の機能を使うことにより、Java のプログラムからネイティブ関数を呼び出すことができます。Java のクラス定義中に、ネイティブ関数を呼び出すメソッドを定義できます。このようなメソッドを「ネイティブメソッド」と呼びます。ネイティブメソッドは、メソッドの定義だけを Java のクラス定義に含めておき、実際に実行されるプログラムはネイティブ関数という具合になります。ネイティブメソッドは、Java のプログラムから見れば、ネイティブ関数への一種のインタフェースということになります。

JNI では、そうした OS 独自の機能やハードウエアを直接見るようなことだけでなく、たとえば、既存のソフトウエア資源をそのまま利用するということにも使えます。

たとえば、ある必要なライブラリが C 言語で作成されているような場合、それを Java に移植するというのも 1 つの手段ですが、C 言語で作られた部分はそのままに、それを Java から利用するようにして、効率的にソフト開発を進めることができるようにするという目的でもネイティブメソッドの呼び出し機能は役立つでしょう。

Mac OS では、Code Fragment Manager 管理下の共有ライブラリを Java のプログラムから呼び出すことができます。Windows では Win32 DLL を利用できます。

もちろん、ネイティブなライブラリは、特定のターゲット OS 上、あるいは場合によっては特定のコンフィグレーションのコンピュータでしか動作しません。JNI を使うということになれば、多くの場合はクロスプラットフォームという Java のメリットは犠牲になるでしょう。それでも、目的を達成するためには JNI は必須ということもあるかもしれません。なお、Java からネイティブメソッドを呼び出すという部分はおおむね共通に作れるので、たとえば、Mac OS 用の共有ライブラリと、Windows 用の DLL をそれぞれ同じ外部仕様で製作しておくことで、Java 側のプログラムはほぼ共通に利用できます。JNI だからといって必ずしも特定のターゲット OS でしか機能しないわけではなく、むしろターゲット OS 間の違いを最大限に吸収するという効果を期待できる場合もあります。

JNI と JDirect2

Java の本来の機能、つまり JDK で定義されている基本機能として、JNI があります。つまり、JNI 自体はクロスプラットフォームで利用できます。呼び出されるネイティブなライブラリの作り方が、ターゲット OS や Java VM による違いというものはありますが、Java で作るプログラム部分は、ターゲットに限らず同じ記述ができます。

一方、MRJ には、JDirect という機能が含まれています。MRJ 2.2 では JDirect2 というバージョンが含まれています。これも JNI と同様、C 言語で作成されたライブラリを呼び出すための機能です。これらの比較を簡単にまとめてみました。

表 12-1　JNI と JDirect2 の比較

特徴	JNI	JDirect2
クロスプラットフォーム	○	×
オーバーヘッド	より多い	より少ない
ネイティブメソッドから Java 側のコントロール	○	×
アプレットでの利用	○	×

JNI で構築したネイティブメソッドの場合、クロスプラットフォームでの運用も不可能ではありません。もちろん、Mac OS のシステムコールを Windows でも動かすことは不可能ですが、入出力がなく計算だけを行うような汎用ライブラリだと、異なる OS をターゲットにしてそれぞれをコンパイルしておくということは可能になるでしょう。その場合、Java 側のプログラムはターゲットごとの違いはほとんどないようにすることもできます。

一方、JDirect2 のドキュメントでは、ネイティブ関数の呼び出しに関わるオーバーヘッドは、JDirect2 の方が少ないと説明されています。パフォーマンスについては JDirect の方がよりかせぐことができるという具合です。

MRJ では、Mac OS 本来の機能を使うために Mac OS のシステムコールを呼び出す必要があります。そうした場面では、クロスプラットフォームということは不要なことなので、パフォーマンスがより良い JDirect を使っているわけです。

Mac OS 上で機能する Java のプログラムは、JNI でも JDirect でもいずれも利用することができるのですが、開発においては選択はしておく必要があるでしょう。ただ、パフォーマンスということは気になりますが、独自に製作するプログラムの場合だと、移植性が高いことやなんと言っても Java の基本機能である JNI を使うということが優先されるのではないかと考えられます。その意味では、JDirect では MJR 自体が Mac OS のシステムコールを使うために用意された仕組みで、応用プログラムでは JNI を使うのが一般的ではないかとも考えられます。しかしながら、応用プログラムで JDirect を使うこともできるので、メリットやデメリットを考えて判断する必要があるでしょう。この章では、JNI を使ったプログラム作成を説明します。JDirect については、MRJ SDK に詳細なドキュメントがあるので、それを参照して下さい。

なお、JNI では、C あるいは C++のプログラムの側から、Java のメソッドを呼び出したり、クラスに定義したインスタンス変数やあるいはクラス変数の値を参照したり設定したりができます。つまり、ネイティブライブラリ側から Java のデータ構造に

アクセスできるようになっています。こうした複雑な機能までも JNI は対応しています。

Java アプリケーションのプロジェクト作成

以下､実際に CodeWarrior Pro4 を使ってごくシンプルなネイティブライブラリを Java のアプリケーションから呼び出すプログラムを作成してみます。完全にこの通りにしないと作れないとうわけではありませんが、作成の流れを手順を追って説明をしておきたいと思います。

まず、CodeWarrior Pro4 の IDE を起動します。「ファイル」メニューの「新規プロジェクト」を選択すると、次のようなダイアログボックスが表示されます。ここでは Java のアプリケーションを作るので「Java」のカテゴリーの下位項目を開き「Java Application」を選んで OK ボタンをクリックします。

図 12-1　Java のアプリケーションのプロジェクトを用意することにした

この後、次のようにダイアログボックスが表示され、プロジェクトのファイル名を指定します。ここでは「フォルダを作成」のチェックボックスが入っているので、指定したフォルダに、指定した名前のフォルダが作成されます。そのフォルダの中に、フォルダと同名のプロジェクトファイルが作成されます。

図 12-2　プロジェクトを入れるフォルダの場所や名前をダイアログボックスで指定

ここまでは通常の Java アプリケーションと変わりません。初期状態では TrivialApplication.java という簡単なアプリケーションのソースプログラムがありますが、以下、そのクラスの main メソッドを利用することにします。

♦Java アプリケーションの設定

Java アプリケーションとして動かすので、そのための設定をまず確認しておきます。「編集」メニューから「Java Application の設定」を選択して、ダイアログボックスで設定を確認します。まず、左のリストで「ターゲットの選択」を選びます。リンカには Java Linker が選択されていますが、ポストリンカとして Java MacOS Post Linker を選択しておき、Finder でダブルクリックして実行可能なアプリケーションを生成するようにします。実際の Java アプリケーションの設定は、Metrowerks Java の環境で行ってもかまわないのですが、共有ライブラリを移動したりしないといけないので、ここでは生成したアプリケーションを起動する方法を中心に説明します。

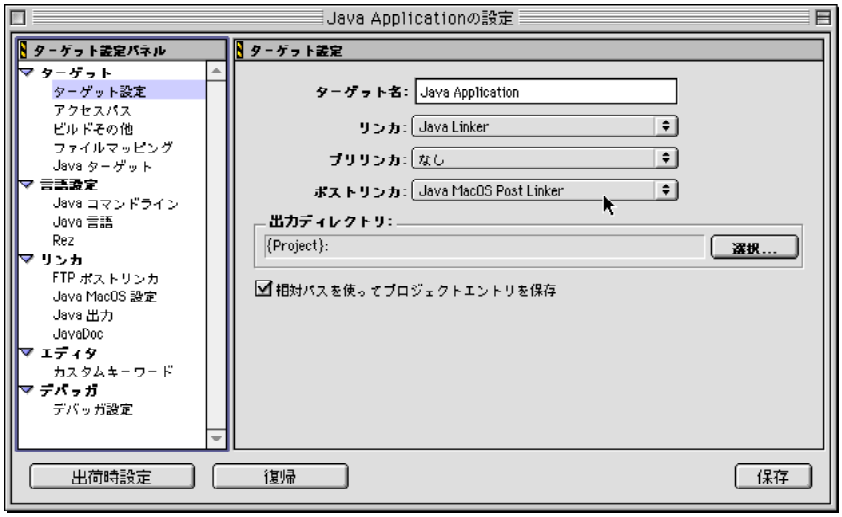

図 12-3　必要ならポストリンカの設定を行っておく

「Java ターゲット」の設定では、起動時に main メソッドを呼び出すクラス名を指定しておきます。TrivialApplication が最初から設定されていますが、main メソッドのあるクラスを変更した場合はここも変更しておきます。バーチャルマシンは通常は Apple MRJ でかまわないでしょう。

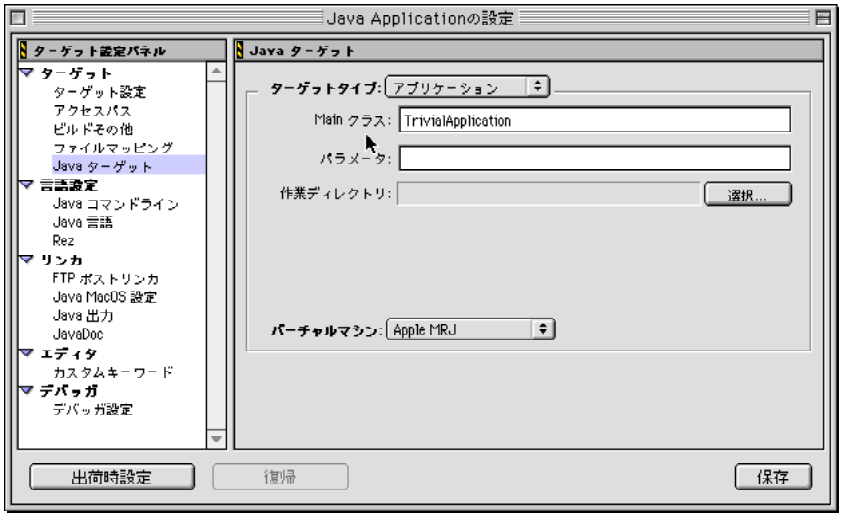

図 12-4　main メソッドのあるクラスを確認する

さらに「Java MacOS 設定」を選択して、生成されるアプリケーションの設定を行います。既定値のままでとりあえず動くことは動きますが、アプリケーションのファイル名やクリエータなどは本来は適切に設定しておく必要があります。

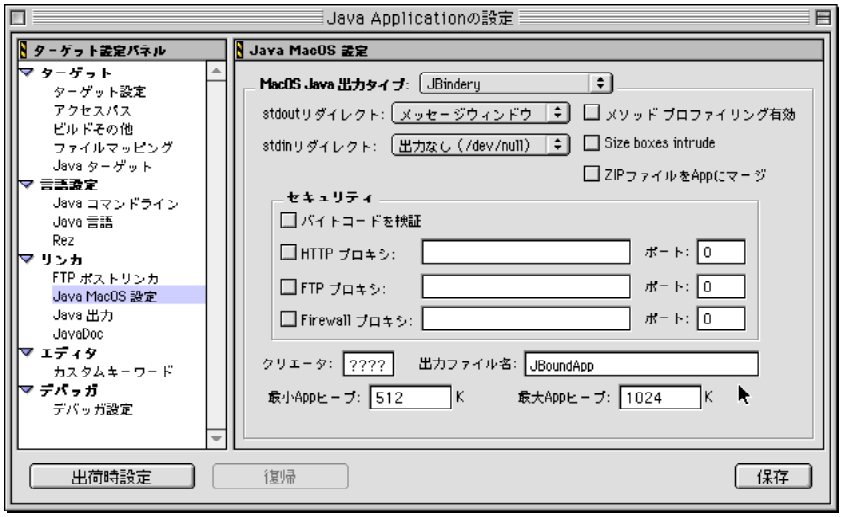

図 12-5　生成アプリケーションに関する設定を確認する

♦ネイティブメソッド用のヘッダを生成する

次に、JNI を利用する場合の独特の設定を行います。「Java Application の設定」のダイアログボックスで、「Java 言語」を選択します。そこにある「ネイティブメソッド用ヘッダーを出力する」というチェックボックスをオンにします。そのグループの 2 つのチェックボックスがありますが、既定値のとおり「ヘッダーにコメントを入れる」はオン、「すべてのクラスのヘッダーを出力する」はオフのままでかまわないでしょう。

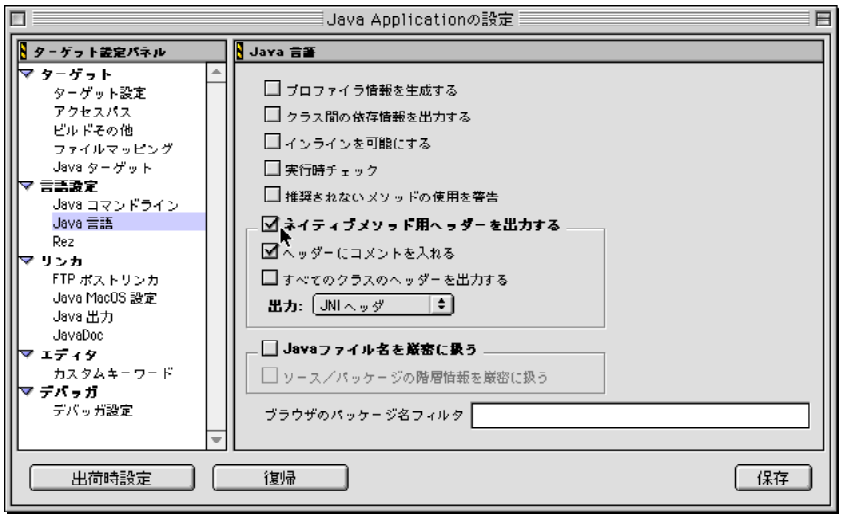

図 12-6　ネイティブメソッド向けのヘッダを生成するようにする

この設定により、Java のソースプログラムをコンパイルしたとき、その中で定義されているネイティブメソッドを認識して、ネイティブ関数のプログラムソースで利用する C あるいは C++で使うヘッダファイルを生成します。

本来、このためのツールは、JDK では javah というものがあります。MRJ SDK でも javah は利用できます。もちろん、これを使ってもいいのですが、CodeWarrior を使えば通常のコンパイル時に javah の処理まであわせて行ってしまうので、非常に便利だというわけです。CodeWarrior を使っていないのであれば、*.java のソースから javah を使ってヘッダーの生成を行って下さい。

♦ネイティブメソッドを定義したクラスを定義する

次に、実際にネイティブ関数を呼び出すネイティブメソッドを定義したクラスを作成します。話を非常に簡単にするために、次のような、1 つのメソッドを呼び出すシンプルなクラスを定義します。「ファイル」メニューから「新規」（Command+N）を選択して、新たにドキュメントウインドウを開きます。そして、以下のプログラムをキータイプします。そして、JNITest.java というファイル名で保存しておきます。

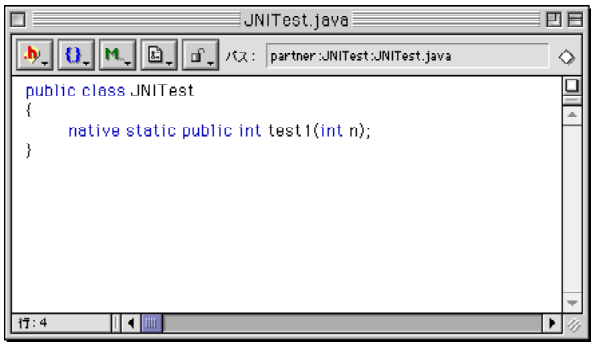

図 12-7　ネイティブメソッドを呼び出すクラスを定義した

ネイティブ関数を利用するには、ネイティブメソッドを定義します。通常のメソッドはクラスに記述したプログラムの実行を行うのに対して、native キーワードを付けたネイティブメソッドは、クラス内ではメソッドの定義だけがあり、実際には関連づけられたネイティブライブラリの方のネイティブ関数を実行するという具合です。関連づけについては、後で説明をします。つまり、ネイティブライブラリにある関数を、Java のメソッドとして公開するのが、nattive キーワードを付けたメソッド定義というわけです。上記では、native キーワードを付けて、test1 というメソッドを定義しています。そして、この test1 に相当するネイティブ関数を、共有ライブラリに仕込むの

です。仕込みもこれから説明します。

♦Java アプリケーション側のプロジェクトとコンパイル

作成した JNITest.java のプログラムのソースウインドウをアクティブにして、「プロジェクト」メニューから「ウインドウの追加」を選択します。そうすると、プロジェクトにソースが追加されます。以下は、追加されたソースを Sources グループにドラッグ&ドロップで移動して、整理したところです。

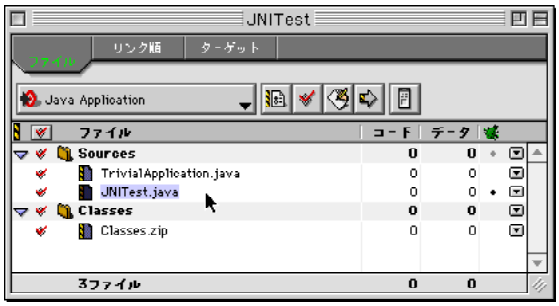

図 12-8　ネイティブメソッドを呼び出すクラスをプロジェクトに追加した

この状態で、まず、完全にコンパイルをします。たとえば「プロジェクト」メニューの「メイク」（Command+M）を実行します。すると、プログラムに間違いがなければ、コンパイル結果のアーカイブ（以下の例では AppClasses.jar）や、アプリケーション（以下の例では JBoundApp）などが生成されます。それに加えて、MWJava Generated Stubs というフォルダが作成されています。その中に、JNITest.h というヘッダファイルが自動的に作成されています。Java 言語の設定でヘッダファイルを生成することを指定したため、このファイルが作られているのです。

図 12-9　Java のアプリケーションをメイクした結果、ヘッダも生成された

まだこの状態ではネイティブライブラリが作られていないので、実行してもネイテ

ィブメソッドからネイティブ関数の呼び出しが行われるわけではありません。

ネイティブライブラリの作成

次にネイティブ関数を含むライブラリを作成します。CodeWarrior では複数のターゲットを 1 つのプロジェクトに作成できるます。JNITest というプロジェクトに新たに Mac OS の共有ライブラリを生成するターゲットを加えることにします。Java Application というのは、最初から用意されている既定のターゲットです。

プロジェクトのウインドウをアクティブにします。そして、プロジェクトウインドウ上部の「ターゲット」と書かれた部分をクリックして、ターゲット一覧を表示しておきます。その状態で、「プロジェクト」メニューから「新規ターゲットを作成」を選択します。

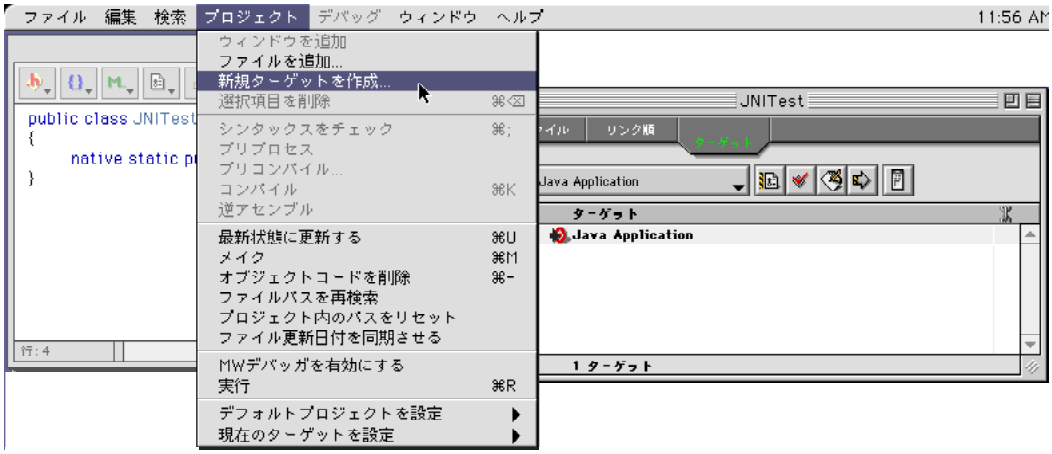

図 12-10　新しくターゲットを作成する

すると、次のようなダイアログボックスが表示されます。ターゲット名は適当に選択します。ラジオボタンは「何も追加しない」を選んでおきます。

図 12-11　新しく作るターゲット名をキータイプする

これで新しいターゲット「Native Library」が作成されました。ターゲット名の左側にはダーツの的と思しき図柄のアイコンがあり、現在選択されているターゲットには→マークが入っています。現在選択されているターゲットを切り替えるには、ターゲットのリストのすぐ上の部分（図では、Java Application となっているグレーのボックス）がポップアップメニューになるので、そこで選択して切り替えることができます。

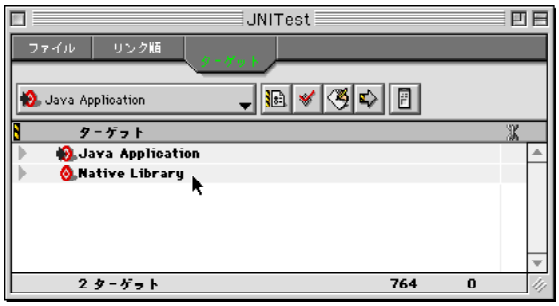

図 12-12　ターゲットが新しく作成された

以下の作業はNative Libraryに対して行うので、ここではターゲットをNative Libararyに切り替えておいてください。

♦ネイティブライブラリで使うヘッダ

ここで、MWJava Generated Stubs というフォルダにある JNITest.h というヘッダファイルを開いてみましょう。Finder 上でダブルクリックするのが早いでしょう。これを開くと、次のように、C/C++で使えるヘッダが出てきます。コメントにあるように、このヘッダは修正しない方が良いでしょう。

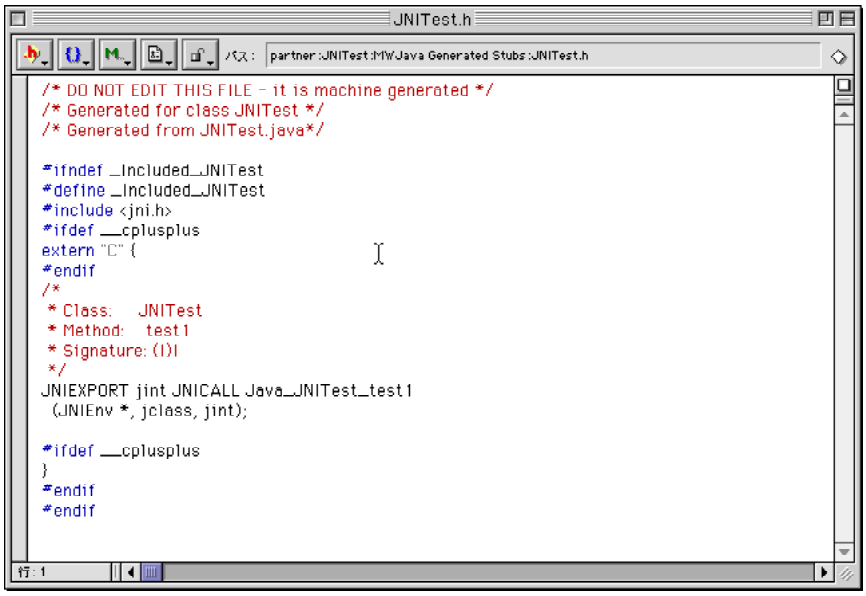

図 12-13　生成されたヘッダファイル

ヘッダファイルの内容は、まず、jni.h というヘッダの読み込みです。このファイルは、CodeWarrior Pro 4 のフォルダの中からたどって、Metrowerks CodeWarrior:Java Support:Metrowerks:Headers:jni.h にあるものを利用します。この jni.h の機能については、12-2 節で説明をします。そして、関数のプロトタイプがあります。コメントを含めて抜き出すと次のようになっています。

```
/*
 * Class:        JNITest
 * Method:       test1
 * Signature: (I)I
 */
JNIEXPORT jint JNICALL Java_JNITest_test1
  (JNIEnv *, jclass, jint);
```

コメントを見ると、つまりこの関数は、Java の JNITest クラスの test1 メソッドの実体を作るものであることがわかります。Signature は引数と戻り値を示す記号で、() 内は引数、() の後に戻り値の型を決められた記号で示します。I は int 型のことです。このプロトタイプをもとにして、これに相当する関数の定義を、C++のソースで作成するわけです。関数名は、Java_JINITest_test1 のように、Java で定義したメソッド名と異なり、Java というキーワードやクラス名をアンダーラインで結んだものです。jint や jclass などは、ネイティブメソッドで使う基本的な型だと考えて下さい。

♦ネイティブライブラリのソースを作成する

今度は C++の側のプログラムを作成します。「ファイル」メニューから「新規」

（Command+N）を選択して、新しくウインドウを作り、以下のプログラムソースを入力します。そして、ここでは NativeCode.cpp という名前でファイルを保存しておきました。なお、ソースはもちろん手入力をしてもいいのですが、関数名の定義や引数あるいは戻り値の定義あたりの部分は、JNTTest.h の該当する部分をコピー&ペーストするのが安全かつ確実で楽な方法です。

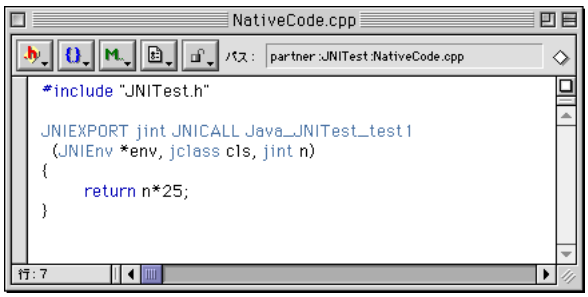

図 12-14　ネイティブメソッドのプログラムを作成する

プログラム自体は、引数に指定した数値を 25 倍して戻すという極めてシンプルなものです。

そして、このプログラムファイルをプロジェクトに追加します。ソースのウインドウがアクティブな状態で、「プロジェクト」メニューから「ファイルの追加」を選択します。すると、この状態ではターゲットがいくつかありますので、どのターゲットに追加するのかを指定するダイアログボックスが表示されます。もちろん、Java の方には追加しないようにして、OK ボタンをクリックします。

図 12-15　ネイティブライブラリのターゲットにだけ C++のソースを追加する

そして、グループとして C++ Sources を新たに作成して、そこに作成した JNITest.cpp のファイルを分類しておきます。グループを作成するには、プロジェクトのウインド

ウをアクティブに、ウインドウ上部の「ファイル」の部分をクリックして一覧をファイル表示にしておき、「プロジェクト」メニューから「新規グループを作成」を選択します。

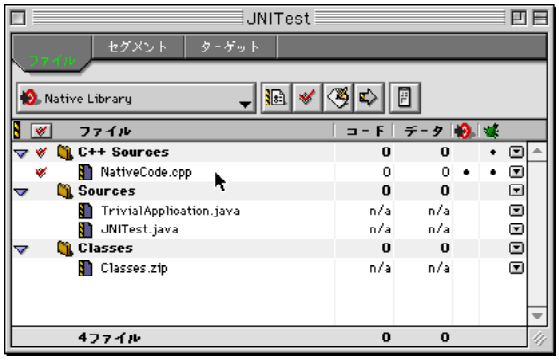

図 12-16　C++**のソースを追加したプロジェクト**

なお、C++のソースの側で Mac OS のシステムコールを呼び出したり、標準ライブラリを使う場合には、該当するライブラリをさらにプロジェクトに追加する必要があります。ここで紹介したプログラムは単に計算しかしないので、これ以上の追加のライブラリは必要ありません。

♦生成ヘッダを参照するようにアクセスパスを設定

ターゲットとして Native Library が選択されているの確認して、「編集」メニューから「Native Library の設定」を選択し、設定を行います。まず、このままでは、JNITest.h ファイルのインクルードができないので、ヘッダファイルのあるフォルダをアクセスパスに追加します。左側のリストで「アクセスパス」を選択します。そして、右側では「ユーザパス」のボックスをクリックして選択し、下の方にある「追加」ボタンをクリックします。

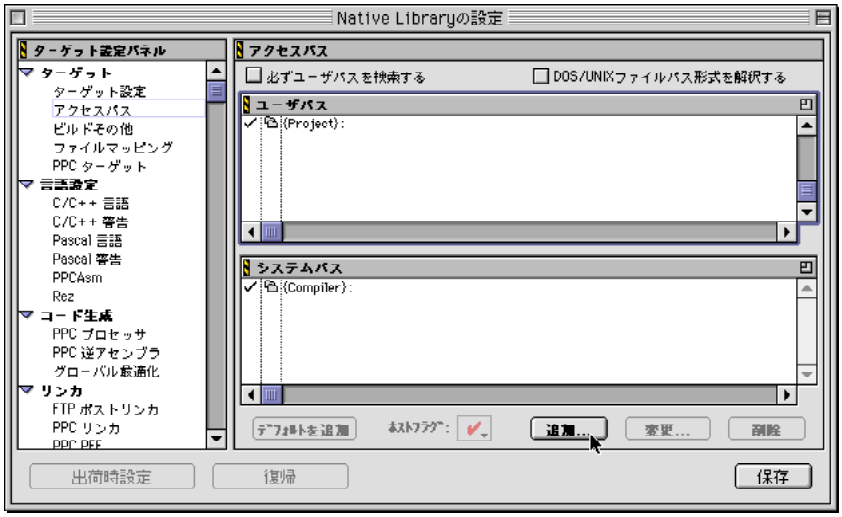

図 12-17　アクセスパスを追加する

次のようなダイアログボックスが表示されるので、生成されたヘッダが保存されている MWJava Generated Stubs フォルダを選択します。また、「プロジェクト相対パス」を指定しておくと、後でフォルダを移動しても設定しなおさずにすみ、便利でしょう。そして、「選択」ボタンをクリックします。

図 12-18　MWJava Generated Stubs フォルダを選択する

すると、ユーザパスに指定したフォルダが追加されます。プロジェクトに相対パスで記録されています。

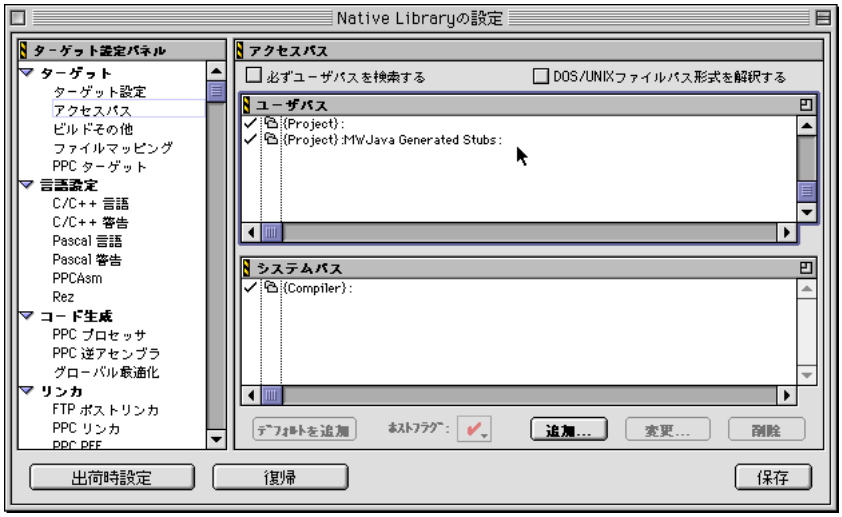

図 12-19　指定したパスがアクセスパスに追加された

♦共有ライブラリを作成する設定

さらに、Netive Library ターゲットの設定を続けます。まず、「ターゲット設定」では、Mac OS の共有ライブラリを作るので、ここはリンカとして「Mac OS PPC Linker」を選択します。

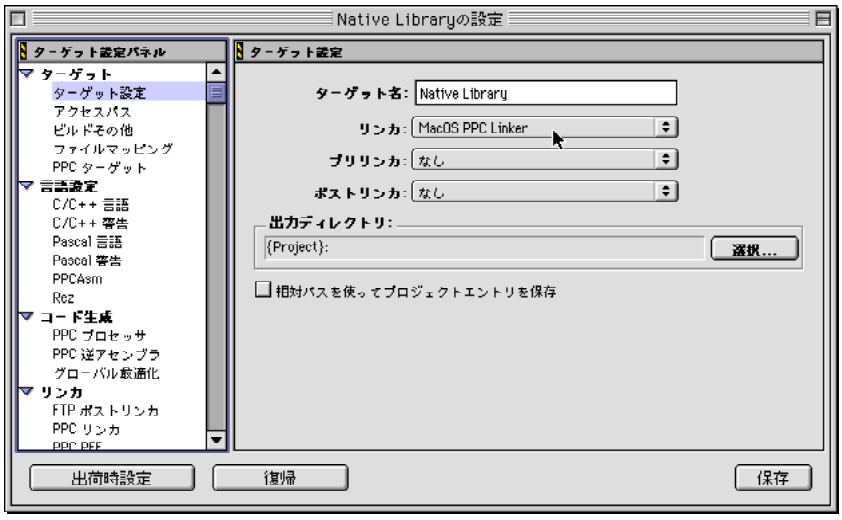

図 12-20　MacOS PPC Linker を選択する

次に「PPC ターゲット」の設定を行います。プロジェクトタイプとしては「共有ライブラリ」を選び、ファイル名は「TestLib」という名前をキー入力しました。クリエータとタイプについては既定値のままでかまいません。ファイル名は任意につけるこ

とができます。

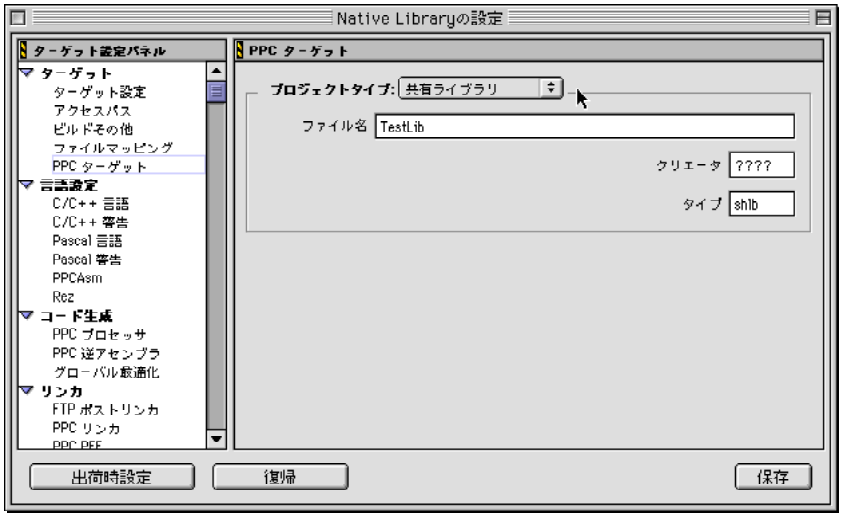

図 12-21　共有ライブラリを生成するようにする

さらに、「PPC リンカ」の設定に移動して、「エントリーポイント」の「メイン」を空欄にしておきます。ここは最初に何か文字列が入っているのが一般的なのですが、共有ライブラリとして使う場合には空欄にしておかないと、リンクエラーが出てきます。

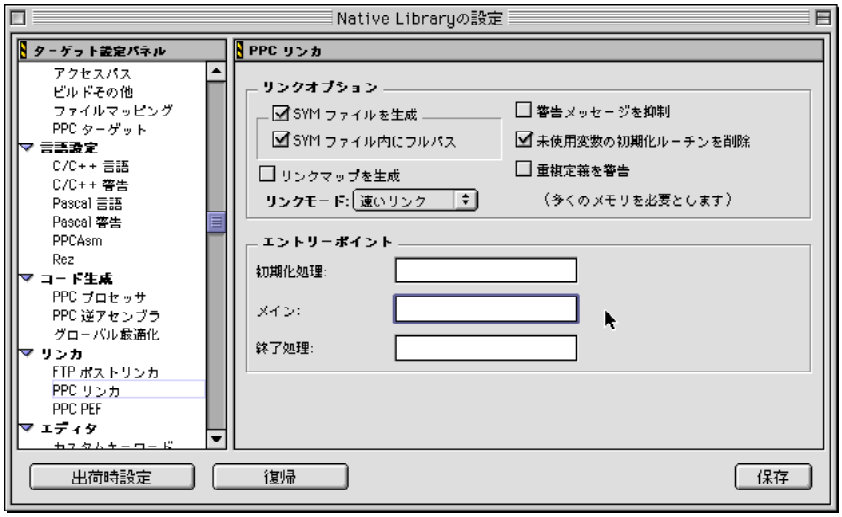

図 12-22　エントリーポイントは削除しておく

さらに「PPC PEF」の設定を行います。まず、「エクスポートシンボル」では「“ .exp ”ファイルを作成」をポップアップメニューから選択しておきます。そして、フラグメ

ント名では「test」と名前を付けました。このフラグメント名を、Java のプログラムで指定する必要があるので、ここでの名前は Java のプログラム側で出てきます。その点を含んで、名前を付けて下さい。

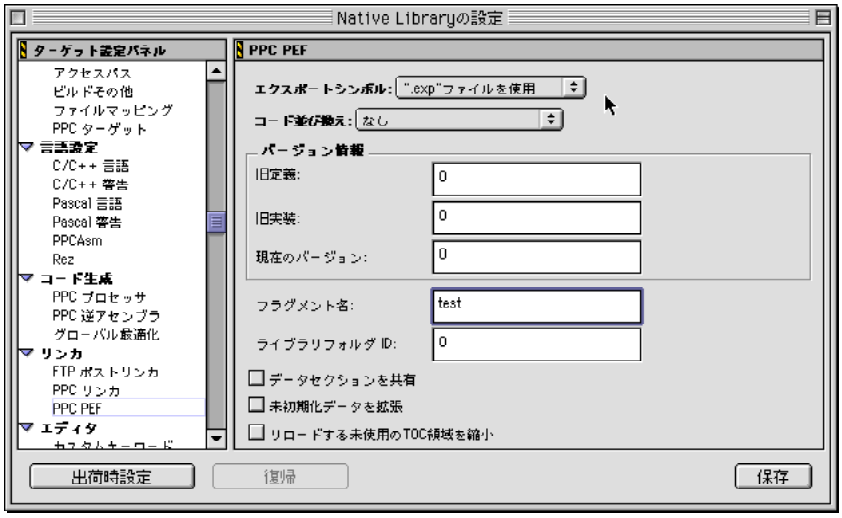

図 12-23　フラグメント名を付け、exp ファイルを出力する

以上のように設定して、「プロジェクト」メニューから「メイク」（Command+M）を選択してメイクをすると、この場合だと、プロジェクトと同じフォルダに、TestLib という名前の共有ライブラリファイルが作成されます。

さらにプロジェクトと同じフォルダに、JNITest.exp というファイルが作られます。このファイルは、中身は「Java_JNITest_test1」という 1 行だけですが、Java のアプリケーションをリンクする段階で使われるるため、このファイルを作っておかないと実行時にエラーが出ます。

図 12-24　共有ライブラリファイルが作成された

共有ライブラリを利用する Java プログラム

再度、ターゲットとして Java Application を選択して、Java のアプリケーション側でネイティブライブラリを利用するようにプログラムを作成します。最初からある TrivialApplication.jara を次の図のように変更しました。

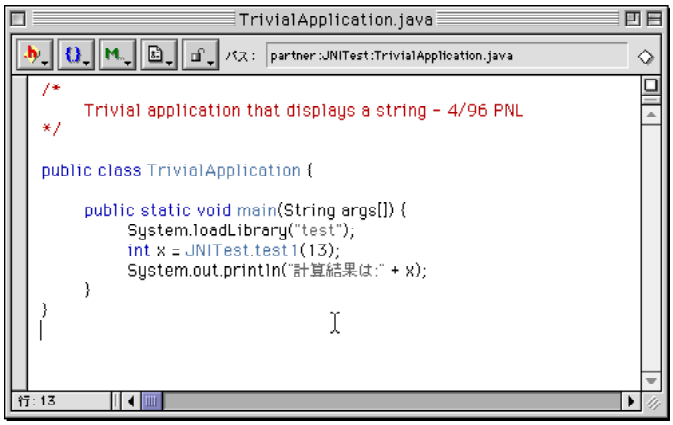

図 12-25　ネイティブライブラリを利用する

プログラムのポイントは、まず、System クラスのクラスメソッドである loadLibrary を利用して、ネイティブのライブラリを Java のアプリケーションで使えるようにするという部分です。ネイティブのライブラリとして、ライブラリのファイル名ではなく、フラグメント名で指定した文字列が指定されているのを注意してください。

こうしてネイティブライブラリを利用できるようにすると、JNITest クラスのクラスメソッドである test1 が使えるようになります。ネイティブメソッドをクラスメソッドにしましたが、必ずしもそうする必要はなく、通常のメソッドで定義してもかまいません。そして、test1 メソッドを実行すると、ネイティブライブラリ側の関数を呼び出して、計算処理をして、13 の 25 倍である 325 を戻しています。

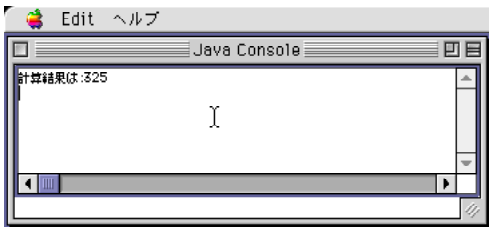

図 12-26　実行結果

12-2 JNI のさまざまな機能

12-1 では、簡単なプログラムで、JNI を利用するためのごく基本的な部分と実際に開発ツールをどう使えばいいかを中心に説明をしました。より複雑なプログラムを作る場合には、特にネイティブメソッドを作る上ではさまざまなサポート機能が利用できます。

JNI についての情報源

JNI については、サンマイクロシステムズのサイトで、仕様書など詳しいドキュメントを参照することができます。また、チュートリアルについても、かなり詳しいものがあるので、情報源としては十分にあると言えるでしょう。以下に、アドレスをまとめておきます。

http://java.sun.com/products/jdk/1.2/docs/guide/jni/index.html

http://java.sun.com/docs/books/tutorial/native1.1/index.html

また、MRJ 環境下での JNI について知っておくべきことが、以下の Technote に記載されています。ある程度 JNI について知識を得た上で、このドキュメントはぜひとも目を通しておいてください。

http://developer.apple.com/technotes/tn/tn1155.html

http://developer.apple.com/ja/technotes/tn1155.html（日本語）

ネイティブライブラリの呼び出し

ネイティブ関数を含むライブラリの呼び出しを行う Java のプログラムは、それほど複雑ではありません。

まず、クラスの中のメソッドを、native というキーワードで定義し、プログラム自体は Java のプログラムでは定義しません。実際に稼動するプログラムは、C/C++で作ったライブラリの中にあります。つまり、Java VM は実行時に、ライブラリの所定の関数を呼び出すという具合に動作するわけです。

native キーワードを付けてメソッドを定義しておき、CodeWarrior で適切な設定を行っていれば、そのネイティブメソッドの C/C++のプログラムで必要になるヘッダファイルが自動的に生成されます。CodeWarrior はその機能が統合されていますが、MRJ

SDK あるいは JDK の javah というツールを使っても同様にヘッダは生成できます。それ以後の作業は C/C++の世界になります。

Java のプログラムの側は、ネイティブメソッドを呼び出すに先立って、Mac OS の場合は共有ライブラリのロードを行う必要があります。そのためのクラスメソッドが System クラスに用意されていて、それをたとえばアプリケーションの起動時などに呼び出すというのが基本的な使い方になります。

表 12-2　共有ライブラリをロードする

戻り値	メソッド	機能
void	System.loadLibrary(String)	引数に指定したフラグメント名の共有ライブラリをロードして利用できるようにする【Static】

共有ライブラリは、Mac OS の場合だと、適当なファイルに保存されていますが、loadLibrary の引数に指定するのはファイル名ではなく、フラグメント名であることは注意してください。共有ライブラリのファイル名は、プログラムの実行にはほとんど関係がありません。

なお、Windows の場合もやはり同様に loadLibrary を使います。Windows では共有ライブラリは、いわゆる Win32 DLL という形式で、拡張子が.dll のファイルを作ります。その場合、loadLibrary の引数には、ファイル名の拡張子を除いた部分を記述するのが一般的です。

12-1 では、static なクラスを定義して、そこにネイティブメソッドを記述しました。従って、クラスのインスタンス化を明示的に行わなくてもメソッド呼び出しはできます。その場合、loadLibrary はプログラムの最初に実行されるように、たとえば、main メソッドなどに入れておくのが基本となるでしょう。もちろん、ネイティブメソッドは、クラスメソッドでないといけないわけではなく、通常のクラスとして new で生成したインスタンスに含まれていてもかまいません。その場合、ライブラリをどこでロードするかということも考慮したいでしょう。たとえば、オブジェクトを new で生成するときにロードしたいということもあるかもしれません。

♦Mac OS での共有ライブラリの場所

Mac OS では、コードフラグメントを含む共有ライブラリのファイルが、アプリケーションと同じフォルダ、あるいはシステムフォルダの「機能拡張」フォルダないしはその 1 階層下までになければなりません。「機能拡張」から 2 階層下のフォルダは検索しないことは注意しておく必要があります。つまり、loadLibrary メソッドは、こ

れらのフォルダだけを検索して、そこにある共有ライブラリファイルのフラグメント名を調べて指定のものがあればロードするという具合です。（なお、アプリケーション自体にコードフラグメントを含めることでもかまいませんが、Java アプリケーションの場合は簡単にはできないでしょう。）したがって、開発を進めているときには、共有ライブラリは Java のアプリケーションと同じフォルダに作り、Java アプリケーションを生成して、それをダブルクリックして起動して動作テストを行うというのが 1 つの方法になるでしょう。

しかしながら、デバッガを使いたいということもあるでしょう。その場合、CodeWarrior では Metrowerks Java というアプリケーションが実行環境のベースとして起動します。そのため、共有ライブラリは、プロジェクトや生成アプリケーションと同じフォルダではなく、Metrowerks Java と同じフォルダに存在する必要があります。そちらにコピーするのもちょっと面倒ですので、このような状況ではむしろ、「機能拡張」フォルダにコピーしておく方がスムーズかもしれません。なお、ファイルのエイリアスがあっても対処できるようなので、場合によってはエイリアスをうまく活用すると良いでしょう。

なお、Metrowerks の Java VM を使う時には、フラグメント名を Java_で始めるなど独特の規則があるので、CodeWarrior のマニュアルやサンプルプログラムなどを参照してください。

共有ファイル自体が正しいフォルダに存在しない場合、loadLibrary で例外が発生し、そのままだとプログラムはストップします。その場合、以下のようなエラーが出てきます。ただし、このエラーは、loadLibrary の引数に存在しないフラグメント名を指定しても同様に出てきます。したがって、共有ライブラリを正しく作ってフォルダに存在するのにこのエラーが出ると言う場合には、フラグメント名が正しく指定されているかもチェックしてくだい。

Exception Occurred:
java.lang.UnsatisfiedLinkError: no test in shared library path

なお、ライブラリファイルがあって、正しくフラグメントの存在が認識できてロードできたとしても、ネイティブメソッドがその中で定義されていないということもあります。その場合は、loadLibrary で次のような例外が発生します。同じ例外ではありますが、メッセージが微妙に違うので、そこで区別します。以下のメッセージは、readResource というネイティブメソッドに対応したプログラムが共有ライブラリに存在しない場合、あるいはそれを正しく認識できない場合に出てきます。

```
Exception Occurred:
java.lang.UnsatisfiedLinkError: readResource
```

ただ、実際に共有ライブラリにあるはずなのに、上記のエラーが出ることがあります。たとえば、共有ライブラリ側の関数を作りたての時に Java アプリケーションを実行するとエラーがでるかと思います。このエラーは、exp ファイルに関係するものです。exp ファイルは単に共有ライブラリを作成した時に作られる関数名の一覧のファイルです。Java のアプリケーションがその exp ファイルを参照して、呼び出すネイティブメソッドを認識しています。そこで、共有ライブラリだけに変更があって再メイクしたときで新しく関数が増えるような場合などでは、Java 側に変更がなくても Java のアプリケーションを再メイクする必要がります。そこで更新された exp ファイルを認識させる必要があるのです。exp ファイルを経由して、共有ライブラリ側の情報を Java のアプリケーションのビルド時に使われることは知っておく必要はあるでしょう。

生成されるヘッダファイル

Java のクラス定義中に、native メソッドを記述すると、そのネイティブメソッドを C/C++言語で記述するのに必要な、ヘッダファイルを生成します。たとえば、12-1 で示した例だと、Java のクラス定義は次のようになっています。ファイル名は、JNITest.java です。

```
public class JNITest
{
    native static public int test1(int n);
}
```

この定義があると、CodeWarrior のヘッダファイル作成機能あるいは javah を使うことで、以下のようなヘッダファイルが生成されます。ファイル名は、JNTTest.h です。

```
/* DO NOT EDIT THIS FILE - it is machine generated */
/* Generated for class JNITest */
/* Generated from JNITest.java*/

#ifndef _Included_JNITest
#define _Included_JNITest
#include <jni.h>
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:       JNITest
```

```
 * Method:       test1
 * Signature: (I)I
 */
JNIEXPORT jint JNICALL Java_JNITest_test1
  (JNIEnv *, jclass, jint);
#ifdef __cplusplus
}
#endif
#endif
```

このヘッダファイルを見ていくことにしましょう。まず、jni.h というヘッダが読み込まれていますが、これは CodeWarrior だと、最初から Java Support フォルダに用意されていて、そこではネイティブメソッドで利用できるさまざまなユーティリティ機能だとか、あるいは基本的な型などの定義が含まれています。jni.h ではネイティブ関数で必要な機能を提供するものだと考えれば良いでしょう。さらに、jni.h から jni_md.h というファイルもインクルードしています。こちらは基本的な型の定義だけです。ネイティブメソッド作成のサポート機能はほとんどが jni.h で定義されています。この機能については、後から説明をしましょう。なお、jni.h と jni_md.h は、MRJ SDK にも含まれていて、MRJSDK:Interfaces&Libraries:CIncludes:という場所にあり、こちらの方が新しいバージョンです。ヘッダ定義の不具合があるようなら、差し換えてみることも検討してください。

そして、コメントでは、JNITest というクラスの test1 というメソッドを実現する関数のプロトタイプを記述しているということが分かります。プロトタイプ宣言では、Java_JNITest_test1 というのが関数名になりますが、命名規則は単に Java でのメソッド名だけでなはく、クラス名や Java というキーワードが付加され、アンダーラインで結び付けられています。そして、引数は 3 つある事が分かります。要は、この通りに実際に関数を定義すればいいということです。この定義をそのままコピーして修正すればいいでしょう。

引数の最初の JNIEnv というクラスは、C++のプログラム中でさまざまなサービスを利用するために使える便利なクラスであると考えれば良いでしょう。使い方の具体例は後から説明しますが、たとえば文字列の処理関数が JNIEnv クラスに定義されているので、プログラムではそれを利用するということになります。

コメント中の Signature は読み方を知っておくと便利かも知れません。これは、引数と戻り値の型を記述したもので、（）内は引数、（）の後には戻り値の型を示す記号を記述します。記号については、表 12-3 を参照してください。この例では、引数が 1 つで int 型なので、（）内は「I」だけになっています。戻り値も int 型なので、「(I)I」

という記述になるわけです。たとえば、「native public long test5(int x, float y);」のようなネイティブメソッドの定義があれば、Signature は「(IF)J」になると言う具合です。文字列などオブジェクトの場合は、L に続いてオブジェクトのパッケージとクラス名を記述します。たとえば、「native public int test(Stirng X):」なら、Signature は「(I)Ljava/lang/String」という具合に記述されます。

ネイティブライブラリの作成

ネイティブ関数を実現するために、C/C++で関数を記述します。ソースプログラムでは、生成されたヘッダ（12-1 の例では JNITest.h）をインクルードすることで、jni.h などに記述されたさまざまな定義が利用できるようになります。言い変えれば、ヘッダをインクルードすることで、ネイティブ側の関数を記述する準備が整ったと言えるでしょう。その段階で、まず、以下のようないくつかの定義型が利用できるようになります。

表 12-3　**基本的な型**

定義されている型	定義元	長さ	Signature	備考
jint	long	32 ビット	I	Java での int 型
jlong	long long	64 ビット	J	Java でのロング型
jbyte	char	8 ビット	B	Java での byte 型
jboolean	unsigned char		Z	Java での boolean 型
jchar	unsigned short	16 ビット	C	Java での char 型。UNICODE なので 16 ビット幅
jshort	short	16 ビット	S	Java での short 型
jfloat	float	32 ビット	F	Java での float 型
jdouble	double	64 ビット	D	Java での double 型
jsize	jint	32 ビット	-	
jobject			L	Java のオブジェクトを参照するクラス
jclass			-	Java のクラスを参照するクラス
jstring			L	文字列を扱うクラス

たとえば、12-1 の例では、Java_JNITest_test1 関数を定義するのですが、その引数は jint 型として定義されています。C/C++のプログラムの中で、本来は Java から来たデータやあるいは Java に送り届けるデータ型として、上記の表のような jint や jstring

が定義されていると考えればよいでしょう。jobject、jclass、jstring は、それぞれ Java のオブジェクト、クラス、そして文字列を扱うためのクラスです。こうした定義があって、型として使えると言う具合に、抽象的に理解しておくことで十分です。

ネイティブメソッドでの文字列の扱い

ネイティブ関数、すなわち C/C++の関数の関数の側で文字列を使う場合には、以下のような JNIEnv クラスのメンバ関数を使用します。

表 12-4　**文字列関連の処理**

戻り値	関数	機能と利用方法
jstring	NewString(const jchar *, jsize)	引数に指定した jchar 型のバッファにあるデータを UNICODE と解釈して、それをもとに jstring 型の文字列を構成する。バッファのいくつ分を取り込むかを引数で指定する
jstring	NewStringUTF(const char *)	引数に指定した char 型バッファ（つまり文字列型）に UTF-8 の文字コードでの文字列が存在するとして、それより jstring 型の文字列を構成する
jsize	GetStringLength(jstring)	jstring 型の文字列の長さを求める
jsize	GetStringUTFLength(jstring)	UTF-8 フォーマットの文字列の長さを求める
const jchar *	GetStringChars(jstring, jboolean *)	jstring 型のデータから jchar 型の配列を得る
const char *	GetStringUTFChars(jstring, jboolean *)	jstirng 型のデータに UTF フォーマットの UNICODE 文字列があるとして、それより char* つまり文字列型のデータを得る
void	ReleaseStringChars(jstring, const jchar *)	文字列のメモリ領域を解放する
void	ReleaseStringUTFChars(jstring, const char*)	文字列のメモリ領域を解放する

たとえば、ネイティブメソッドで文字列が引数にあるとすると、次のような形式で、関数のプロトタイプが作成されます。この場合、戻り値も文字列です。（プログラム

は抜粋です。すべてのプログラムは 12-3 を御覧ください。）

クラス定義：

```
public class StringResource      {
    native private String readResource
          (short vRefNum, int parID, String name, int ID);      //ネイティブコールの定義
}
```

ネイティブメソッドの関数：

```
JNIEXPORT jstring JNICALL Java_StringResource_readResource
  (JNIEnv *ev, jobject obj, jshort vRefNum, jint parID, jstring name, jint ID)
```

つまり、Java からやってくる文字列や、あるいは Java に戻す文字列は、jstring 型でなければなりません。実際に C/C++で文字列処理をするには、jstring 型を char*などの配列に展開しなければなりません。そこで、たとえば引数で得られた文字列は、GetStringUTFChars 関数を使って、char *型のデータに変換します。そうすることによって、C/C++での文字列処理ができるようになります。

```
const char * fName = ev->GetStringUTFChars(name, &isCopy);
```

ここで、Java では文字コードは UNICODE である点を思い出して下さい。Mac OS では Shift-JIS が文字コードなので、文字コードの不一致があります。GetStringUTFChars は、Java の文字列を UTF-8 フォーマットの UNICODE 文字列で戻します。したがって、戻される char*型配列の中身も、UNICODE ではあります。ただし、UTF-8 なので、7 ビット長までのアルファベットや数字など（半角文字）については、ASCII コードとまったく同一です。Java の側で「ResourceFile.rsrc」という文字列が設定されていたとしますと、GetStringUTFChars によって得られた文字列は、ASCII コードで「ResourceFile.rsrc」というように得られます。

このように、半角文字だけを扱うには、GetStringUTFChars を使うのが手軽ですが、日本語の文字を扱うには、UTF-8 を Shift-JIS へ変換する必要が出てきます。12-3 で示すプログラムでは変換をしないで動かしてみることで、どういった不具合が出てくるのかを検証してみましょう。

文字列を利用した場合、ReleaseStringUTF などの関数で、メモリを開放しておく必要があります。これをしないと、メモリリークの原因になるとされています。この関数は、jstring と char*の 2 つの引数を指定しますが、たとえば、GetStringUTFChars で文字列を作った場合には、そこで引数に指定した jstring と、GetStringUTFChars 関数の戻り値として得られた char*を指定すれば良いでしょう。

逆に、C/C++で得られた文字列を、ネイティブ関数の戻り値として戻したい場合も

あります。その場合、定義にあるように、jstring 型で戻さなければなりませんが、char* 型のデータから、jstring 型のデータを作るのが、NewStringUTF という JNIEnv クラスで定義された関数です。たとえば次のように使います。

```
jstring retStr = ev->NewStringUTF((const char *)converted);  //戻り値の文字列を生成する
return(retStr);  //値を返す
```

ここでも、もちろん、戻す文字列は UNICODE でなければなりません。12-3 で示すサンプルプログラムは、リソースファイルのリソースを取り込みますが、その文字列は Shift-JIS です。それをそのまま NewStringUTF を通しても UNICODE にはなりません。NewStringUTF は、「UNICODE で構成された文字列のバイト列を jstring 型に変換する」という機能しか持ち得ていないため、具体的にはあらかじめ UTF-8 フォーマットの文字列に変換しておく必要があります。

12-3 のサンプルでは、こうした文字列変換を、Text Encoding Converter を使って実現しました。もちろん、Mac OS でないと利用できない機能ですが、それ以前に「リソースの読み込み」自体も Mac OS 独自の機能なので、その意味では問題はないでしょう。しかしながら、ネイティブライブラリについてもプラットフォームの移植性を考慮しなければならないのであれば、それに応じた文字列変換の手法を利用することになります。

ネイティブメソッドでの配列の扱い

引数や戻り値に配列を指定することもできます。配列は、C/C++側のプログラムでは、jarray という抽象的なクラスで扱われますが、データの型ごとに、jbooleanArray、jbyteArray、jcharArray、jshortArray、jintArray、jlongArray、jfloatArray、jdoubleArray、jobjectArray の各クラスが定義されています。

以下のような関数が JNIEnv クラスで定義されています。データのタイプごとに関数が定義されているため、それをすべてリストにすることはしません。たとえば、NewBooleanArray という関数があり、その戻り値は jbooleanArray であると読み取って下さい。

表 12-5　ネイティブプログラム側で配列を扱う関数

戻り値	関数	機能と利用方法
jsize	GetArrayLength(jarray)	配列の要素数を求める
j[タイプ]Array	New/タイプ/Array(jsize)	指定したタイプの配列を生成する。引数で要素数を指定する
j[タイプ]*	Get/タイプ/ArrayElements(j[タイプ]Array, jboolean *)	配列オブジェクトから配列に変換する
void	Release/タイプ/ArrayElements(j[タイプ]Array, j[タイプ] *, jint)	配列の要素をリリースする
void	Get/タイプ/ArrayRegion(j[タイプ]Array, jsize, jsize, j[タイプ] *)	配列の内容をいくつか取り出す
void	Set/タイプ/ArrayRegion(j[タイプ]Array, jsize, jsize, j[タイプ] *)	配列の内容をいくつか設定する
jobjectArray	NewObjectArray(jsize, jclass, jobject)	オブジェクトの配列クラスを生成する
jobject	GetObjectArrayElement(jobjectArray, jsize)	オブジェクト配列の要素を取り出す
void	SetObjectArrayElement(jobjectArray, jsize, jobject)	オブジェクト配列の要素を設定する

/タイプ/：Boolean、Byte、Char、Short、Int、Long、Float、Double

[タイプ]：boolean、byte、char、short、int、long、float、double

たとえば、ネイティブメソッドで int 型の配列を指定すると、C/C++側の関数では、引数に jarrayInt 型の変数が並びます。そこで、GetIntArrayElements 関数を使うと jint 型の配列が得られるというわけで、配列の要素にアクセスができるという具合です。

ネイティブメソッド内から Java のクラスを利用する

ネイティブメソッド内から、Java のインスタンス変数を取り出したりあるいは設定したり、さらには Java のメソッドの呼び出しもできます。そのための基本的なデータを扱う型として、jfieldID、jmethodID が定義されています。このような型があると抽象的に理解しておくのでかまいません。jfieldID はインスタンス変数の指定、jmethodID はメソッドの指定に利用します。インスタンス変数を利用するための機能として、以下のような関数が JNIEnv クラスに定義されています。データのタイプご

とに関数が定義されていますが、たとえば、Java クラスのインスタンス変数の値を int 型として取り出す関数は、GetIntField で、その戻り値は jint 型であるという具合です。

表 12-6　インスタンス変数やクラス変数の処理

戻り値	関数	機能と使用方法
jclass	GetObjectClass(jobject)	オブジェクトのクラスを得る
jfieldID	GetFieldID(jclass, const char *, const char *)	フィールド ID を取得する。引数には順番に、対象クラス、インスタンス変数の名前、型を示すシグネチャを指定する
j[タイプ]	Get/タイプ/Field(jobject, jfieldID)	インスタンス変数の値を取得する。引数には順に対象オブジェクト、フィールド ID を指定する
void	Set/タイプ/Filed(jobject, jfieldID, j[タイプ])	インスタンス変数に値を設定する。引数は順に、対象オブジェクト、フィールド ID、設定する値を指定する
j[タイプ]	GetStatic/タイプ/Field(jclass, jfieldID)	クラス変数の値を取得する。引数には順に対象クラス、フィールド ID を指定する
void	SetStatic/タイプ/Filed(jclass, jfieldID, j[タイプ])	クラス変数に値を設定する。引数は順に、対象クラス、フィールド ID、設定する値を指定する

/タイプ/：Object、Boolean、Byte、Char、Short、Int、Long、Float、Double、Void

[タイプ]：object、boolean、byte、char、short、int、long、float、double、void

まず、ネイティブ関数では、引数で、呼び出し側のクラスが引き渡されます。その値を利用して、GetObjectClass 関数を使い、jclass 型のデータを得ます。さらに、GetFieldID を利用して、指定した名前のインスタンス変数のフィールド ID を取得します。その後、変数の値を読み出したり設定することを行いますが、どのインスタンス変数なのかはフィールド ID で指定するという具合です。

```
JNIEXPORT jstring JNICALL Java_StringResource_readResource
  (JNIEnv *ev, jobject obj, jshort vRefNum, jint parID, jstring name, jint ID)
{
    jboolean isCopy;

    jclass clazz = ev->GetObjectClass(obj); //呼び出し元クラスを得る
    jfieldID fieldID = ev->GetFieldID(clazz, "err", "I");
          //呼び出し元クラスのインスタンス変数 err のフィールド ID を得る

    ev->SetIntField (obj, fieldID, -1);    //呼び出し元クラスの err に-1 を書き込む

          :
```

以上のプログラムの呼び出し元のクラスに err という int 型のインスタンス変数が定義されています。そのインスタンス変数に、SetIntField で、-1 という値を書き込んでいるわけです。GetFieldID では、インスタンス変数名を文字列で指定するとともに、その型を Signature の記号として文字列で指定します。この指定を間違えると、正しく値が設定されなくなるので、注意する必要があります。また、Get__や Set__関連の関数もデータの型に応じたものが用意されているので、基本的にはインスタンス変数の型と合わせる必要があります。

メソッドの呼び出しでは以下のような関数が、JNIEnv クラスに用意されています。引数の渡し方として、書き並べる方法と、va_list 型を利用する方法、配列を利用する方法の 3 つが用意されています。それぞれ、戻り値のデータの型ごとに関数が用意されているので、jni.h ファイルでの定義はたくさんあります。基本的な使い方はインスタンス変数と同様で、GetOjcectClass で jclass を得て、GetMethodID でメソッド ID を取得します。そのメソッド ID で呼び出すメソッドを指定します。/タイプ/と[タイプ]で使うキーワードについては表 12-6 を参照してください。

表 12-7　ネイティブメソッドから Java のメソッドを呼び出す

戻り値	関数	機能と利用方法
jmethodID	GetMethodID(jclas, const char *, const char *)	メソッド ID を取得する。引数には順番に対象クラス、メソッドの名前、メソッド呼び出しのシグネチャを指定する
j[タイプ]	Call/タイプ/Method(jobject, jmethodID, ...)	メソッドを呼び出す
j[タイプ]	Call/タイプ/MethodV(jobject, jmethodID, va_list)	メソッドを呼び出す
j[タイプ]	Call/タイプ/MethodA(jobject, jmethodID, jvalue *)	メソッドを呼び出す
j[タイプ]	CallNonvirtual/タイプ/Method(jobject, jclass, jmethodID, ...)	スーパークラスのメソッドを呼び出す
j[タイプ]	CallNonvirtual/タイプ/MethodV(jobject, jclass, jmethodID, va_list)	スーパークラスのメソッドを呼び出す
j[タイプ]	CallNonvirtual/タイプ/MethodA(jobject, jmethodID, jvalue *)	スーパークラスのメソッドを呼び出す
j[タイプ]	CallStatic/タイプ/Method(jclass, jmethodID, ...)	クラスメソッドを呼び出す
j[タイプ]	CallStatic/タイプ/MethodV(jclass, jmethodID, va_list)	クラスメソッドを呼び出す
j[タイプ]	CallStatic/タイプ/MethodA(jclass,	クラスメソッドを呼び出す

```
                    jmethodID,    jvalue *)
```

12-3 サンプルプログラム

リソースファイルから STR リソースを読み取るクラスを作ってみました。もちろん、Resource Manager の機能の呼び出しなどで、共有ライブラリを使っています。12-1 で紹介したプログラムと併せて、まとめておきましょう。

プログラムの実行例

プログラムの実行結果はコンソールに出力するようにしておきました。出力される最初の 1 行は、12-1 で紹介した部分です。残りはリソースファイルの STR リソースを読み出した結果です。

プロジェクトと同じフォルダに、ID=128 と 129 の 2 つの STR リソースが含まれたファイル ResourceFile.rsrc を作成しておきました。そして、ID=128 の方は日本語の文字列が入っています。その結果を読み出したのが「あいうえお」です。頭の数字は ResError 関数の戻り値です。ID=129 のリソースには English String という文字列が入っています。Resource ERROR という文字列は、存在しないリソースを読み込もうとした時に、エラーメッセージとして戻したものです。さらに、Nothing というリソースファイルを開こうとしますが、それは存在しないため、エラーコードとして-1 を戻しています。また、「日本語ファイル.rsrc」というファイルは存在するのですが、ここで作成したプログラムでは日本語文字列を Java 側からネイティブ関数に引き渡したままで扱っているため、ファイルがないと思ってエラーを出しています。つまり、UTF-8 のままファイル名の文字列を扱っており、Resource Manager の API コールでは Shift-JIS での文字列を必要とするため、文字コードの不一致が生じてトラブルになるわけです。

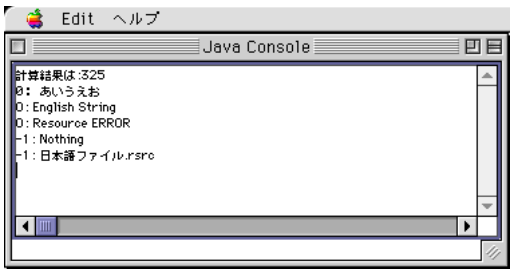

図 12-27 サンプルプログラムの実行例

なお、プロジェクトにはいろいろなファイルを追加しなくてはなりません。まず、

リソースファイルの vRefNum など、HFS でのパラメータを得るために、MRJClasses.zip に定義が含まれている FSSpec クラスを利用しています。File オブジェクトから Mac OS のシステムで使うパラメータへの分解は Java の側で行っているというわけです。

また、ネイティブライブラリの方では Mac OS の機能を使っているために、InterfaceLib はプロジェクトに含めなければなりません。また、Text Encoding Converter の機能を使うためにこれも適切なライブラリを含めなければなりません。」しかしながら、それに相当するものが CodeWarrior のフォルダには見当たらなかったので、システムフォルダの「機能拡張」フォルダにある「Text Encoding Converter」を直接プロジェクトにインクルードしました。また、CodeWarrior Pro 4:Metrowerks CodeWarrior:MacOS Support:Libraries:Runtime:Runtime PPC にある、MSL ShLibRuntime.Lib というファイルも加えないとリンクエラーが出てしまいます。いずれにしても、このあたりは、Mac OS のプログラムとして完結させるために必要な措置で、Java や JNI とはほとんど関係がないところではあります。

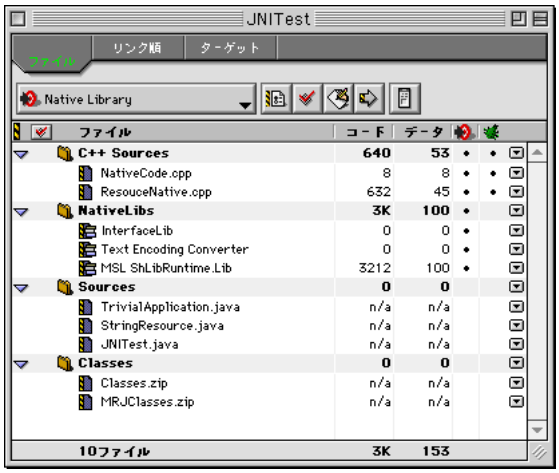

図 12-28　プロジェクトの構成

なお、プログラムは PowerPC だけでしか機能しません。共有ライブラリをそのように作ってあります。CodeWarrior のサンプルを見ると、PowerPC や 68k の両方で動くように作ってあるなど複雑な構成になっています。ただ、現状の MRJ 2.1 が PowerPC 上でしか機能しないことを考えれば、現段階で作るプログラムは 68k への対応は不要だと考えます。

TrivialApplication.java

main メソッドのあるクラスがこのクラスですが、CodeWarrior の Java Application のテンプレートをそのまま使っています。ネイティブメソッドのある JNITest クラスと、StringResource というクラスをインスタンス化してそれにあるメソッドを呼び出しています。

```
import java.io.*;
import java.util.*;
import java.text.*;

public class TrivialApplication    {

    public static void main(String args[])
    {
        System.loadLibrary("test");
        int x = JNITest.test1(13);
        System.out.println("計算結果は:" + x);

        String curDPath = System.getProperty("user.dir");   //アプリケーションのフォルダを得る

        StringResource sr
            = new StringResource(new File(curDPath,"ResourceFile.rsrc"));
                //アプリケーションと同じフォルダにある ResourceFile.rsrc という
                //ファイルの STR  リソースを読む準備を行う
        String rstr = sr.readStringResource(128);       //ID=128 のリソースを読む
        System.out.println(sr.rsrcError() + ": " + rstr);
                                        //エラーとともに読んだリソースを表示（以下同）
        rstr = sr.readStringResource(129);
        System.out.println(sr.rsrcError() + ": " + rstr);
        rstr = sr.readStringResource(12345);
        System.out.println(sr.rsrcError() + ": " + rstr);

        sr = new StringResource(new File(curDPath,"Nothing"));
                                        //Nothing というファイルは存在しない
        rstr = sr.readStringResource(128);
        System.out.println(sr.rsrcError() + ": " + rstr);

        sr = new StringResource(new File(curDPath,"日本語ファイル.rsrc"));
            //日本語ファイル.rsrc というファイルは存在するが、
            //ネイティブプログラム側では日本語のファイル名には対応していない
        rstr = sr.readStringResource(128);
        System.out.println(sr.rsrcError() + ": " + rstr);
    }
}
```

JNITest.java

12-1 で紹介した、単にネイティブメソッドを呼び出すだけの単純なクラスです。

```
public class JNITest
{
     native static public int test1(int n);
}
```

JNITest.h

JNITest.java をコンパイルする時にヘッダを生成するようにすれば、以下のようなヘッダを自動的に生成します。

```
/* DO NOT EDIT THIS FILE - it is machine generated */
/* Generated for class JNITest */
/* Generated from JNITest.java*/

#ifndef _Included_JNITest
#define _Included_JNITest
#include <jni.h>
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:         JNITest
 * Method:       test1
 * Signature: (I)I
 */
JNIEXPORT jint JNICALL Java_JNITest_test1
   (JNIEnv *, jclass, jint);

#ifdef __cplusplus
}
#endif
#endif
```

NativeCode.cpp

自動的に生成されたヘッダをもとに、ネイティブ関数を、C++の関数として作成しました。プログラム内容は、単に引数の値を 25 倍するだけのものです。

```
#include "JNITest.h"
```

```
JNIEXPORT jint JNICALL Java_JNITest_test1
  (JNIEnv *env, jclass cls, jint n)
{
    return n*25;
}
```

StringResource.java

ファイルの中にある STR リソースから ID 番号を指定してリソースを読み取り、その内容の文字列を得るクラスです。リソースファイルの File オブジェクトを引数にコンストラクタを呼び出します。コンストラクタでは、リソースファイルを引数に取ります。そして、MRJ にある FSSpec クラスの機能を使って、File Manager で使う vRefNum、parID、name 式のファイル指定パラメータを求め、それらをインスタンス変数に残します。

実際にリソースを読み出すのが readStringResource メソッドで、引数には ID 番号だけを指定します。読み取った結果は、戻り値として戻ります。このメソッドの中で、ネイティブなメソッドを呼び出し、そちらで Resource Manager のシステムコールを使って実際にリソースの読み込み処理を行っています。

```
import java.io.*;
import com.apple.mrj.macos.toolbox.FSSpec;

//   STR リソースからテキストを読み出すクラス
//   内部的に、Toolbox コールを利用している

public class StringResource     {

    private File resFile;     //File オブジェクトでのリソースファイル
    private short vRefNum;       //リソースファイルの FSSpec での各メンバー
    private int parID;
    private String name;

    public int err;     //リソースエラーを記録する変数

    public StringResource(File f)       {      //コンストラクタ、引数にリソースファイルを指定
        resFile = f;         //インスタンス変数に記録する
        System.loadLibrary("test");  //ネイティブコードのある共有ライブラリを呼び出す
        FSSpec fspec = new FSSpec(resFile);  //リソースファイルの FSSpec を得る
        vRefNum = fspec.getVRefNum();  //FSSpec での各メンバの値を変数に記録
        parID = fspec.getParID();
        name = fspec.getName();
    }

    public String readStringResource(int ID){      //引数に指定した ID の STR リソースを読み出す
```

```
        err = 0;      //エラーコードをクリア
        String val = readResource(vRefNum, parID, name, ID);    //ネイティブコール
        return val;  //結果を戻す
    }

    native private String readResource
        (short vRefNum, int parID, String name, int ID);       //ネイティブコールの定義

    public int rsrcError()    {
        return err;  //エラーを戻す
    }
}
```

StringResource.h

StringResource.java をコンパイルした時に自動的に生成されるヘッダです。ネイティブメソッドは 1 つだけしか定義されていないので、それに対応したネイティブ関数を定義するプロトタイプがあるだけです。

```
/* DO NOT EDIT THIS FILE - it is machine generated */
/* Generated for class StringResource */
/* Generated from StringResource.java*/

#ifndef _Included_StringResource
#define _Included_StringResource
#include <jni.h>
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:         StringResource
 * Method:       readResource
 * Signature: (SILjava/lang/String;I)Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL Java_StringResource_readResource
  (JNIEnv *, jobject, jshort, jint, jstring, jint);

#ifdef __cplusplus
}
#endif
#endif
```

ResouceNative.cpp

ネイティブメソッドの実際の処理部分、すなわちネイティブ関数のプログラムです。

引数で得られた文字列から char*型にし、さらに Pascal の文字列に変換して、リソースファイルを開くシステムコールに渡しています。また、リソースとして得られた結果は、Text Encoding Converter の機能を使って UTF-8 に変換し、戻しています。さらに、呼び出し元の StringResource クラスにある err というインスタンス変数の値を、このネイティブメソッド内部で書き換えて、リソースエラーを呼び出し元のクラスで認識できるようにしました。なお、Get1Resource はリソースが存在しない場合でも、ResError の結果は 0 になります。Get1Resource の戻り値が nil であれば、リソースが存在せず何もロードしなかったと判断できるので、そのエラー処理も含まれています。

```
#include "StringResource.h"
#include "TextEncodingConverter.h"

JNIEXPORT jstring JNICALL Java_StringResource_readResource
  (JNIEnv *ev, jobject obj, jshort vRefNum, jint parID, jstring name, jint ID)
{
    jboolean isCopy;

    jclass clazz = ev->GetObjectClass(obj); //呼び出し元クラスを得る
    jfieldID fieldID = ev->GetFieldID(clazz, "err", "I");
                                //呼び出し元クラスのインスタンス変数 err のフィールド ID を得
る

    const char * fName = ev->GetStringUTFChars(name, &isCopy);
                                //引数から文字列を得る。文字列は UTF-8 として得られる
    c2pstr((char *)fName); //Pascal 文字列に変換する
    int refNum = HOpenResFile(vRefNum, parID, (unsigned char *)fName, fsRdPerm);
                                //引数の情報をてがかりにリソースファイルを開く
    if(refNum == -1) {      //リソースファイルを開けなかったとき
        ev->SetIntField(obj, fieldID, -1);    //呼び出し元クラスの err に-1 を書き込む
        p2cstr((unsigned char *)fName);   //ファイル名を再度 C 文字列に戻す
        return(ev->NewStringUTF(fName));      //戻り値は開けなかったファイルのファイル名
    }
    ev->ReleaseStringUTFChars(name,   fName);       //使用し終わった文字列をリリースする

    UseResFile(refNum);
    Handle sh = Get1Resource('STR ', ID);  //引数に指定した ID 番号の STR  リソースを得る

    OSErr re = ResError();//リソース関連のエラー番号を得る
    ev->SetIntField(obj, fieldID, (short)re);
                               //それを呼び出し元オブジェクトの err インスタンス変数に書き込む
    if((re != noErr)||(sh == nil))   {      //エラーがある場合
        return(ev->NewStringUTF("Resource ERROR"));    //処理を終える。戻り値はこの文字列
    }

    HLock(sh); //取り出したリソースのハンドルをロックする

    unsigned long strLen = *((unsigned char *)(*sh));
                                    //1 バイト目が文字列のバイト数になっている
```

```
unsigned char converted[512];      //UNICODE 変換した結果を保存する配列を用意
for(int i=0; i<512 ; i++) converted[i] = 0; //保存する配列を初期化（ちょっと非効率的）

/*以下、TextEncodingConverter の機能を使って、Shift JIS の文字列を UTF-8 に変更する*/
TECObjectRef ec;
ByteCount actualInputLength, actualOutputLength;

TextEncoding outputEncoding
      = CreateTextEncoding(kTextEncodingISO10646_1993, 0, kUnicodeUTF8Format);
            //変換後の文字列は、UNICODE で、UTF-8 のフォーマット
      TextEncoding inputEncoding
            =          CreateTextEncoding(kTextEncodingMacJapanese,
kJapaneseStandardVariant, 0);
                  //変換前の文字列は、MacJapanese。すなわち Shift-JIS
      TECCreateConverter(&ec, inputEncoding, outputEncoding);
                  //コンバータのオブジェクトを用意
      TECConvertText(ec, (unsigned char*)(*sh+1), strLen, &actualInputLength,
                        converted, 512, &actualOutputLength);
                  //実際に変換を行う。STR リソースの文字列の UTF-8
形式のバイト列が得られる

      jstring retStr = ev->NewStringUTF((const char *)converted); //戻り値の文字
列を生成する
      HUnlock(sh);      //ハンドルのロックを解除

      CloseResFile(refNum);     //リソースファイルを閉じる
      return(retStr);      //値を返す
}
```

（第 12 章以上）

第13章 RMI の利用

- ネットワーク越しにあるオブジェクトのメソッド呼び出しを行うのが RMI です。
- 分散オブジェクトやクライアント/サーバー式のシステム構築のベースになる機能です。
- Mac OS 上での RMI のクライアントとサーバーの作成方法を説明します。
- CodeWarrior Pro5、MRJ 2.1.4、MRJ SDK 2.1 をベースに解説を行います。

13-1 RMI と分散オブジェクト

ネットワークの先にあるクラスのインスタンスを参照し、そのメソッドを呼び出すメカニズムとして、JDK には、RMI が用意されています。その RMI についての基本的なことをまとめておきましょう。

RMI によるリモート呼び出し

RMI（Remote Method Invocation）は、ネットワークの先にあるクラスのインスタンスに存在するメソッドの呼び出しを行う機能です。JDK で定義されている Java の基本機能で、プラットフォームに依存せずに利用できます。

通常、クラスで定義されたメソッドは、メソッド名をプログラムに記述することで呼び出すことができます。Java では実行時にクラスを探して呼び出し処理を行うというダイナミックバインディングが大きな特徴になっています。RMI を利用すると、一定の準備さえすれば、ネットワーク越しに存在する別のコンピュータに存在するクラスのインスタンスを参照し、あたかもいっしょにコンパイルして用意されているクラスのメソッドと同じように、そのクラスのメソッドを呼び出すことができるようになります。つまり、メソッドを呼び出すと、実際にメソッドの処理を行うのは別のコンピュータであるような使い方ができるのです。

通常のメソッド呼び出しでは、同一マシン内で処理がなされ、メモリ空間も基本的には同じものを使います。一方、ネットワーク越しにメソッド呼び出しを行うとすると、たとえば引数をネットワークを通じて実行するメソッドに送付し、異なるコンピュータ内で処理がなされて、得られた結果は再度ネットワークを通じて呼び出し元に送られるなど、複雑なネットワーク処理が必要なことは容易に分かります。RMI を使うと、こうした複雑なことはほとんどの部分をシステム側で行います。引数や返り値のやりとりでは基本データ型だけでなく、オブジェクトも可能ですが、こうしたことが実現する基盤として、シリアライゼーションの機能も利用しています。

具体的には、呼び出し元のクライアントと、呼び出されるサーバーという関係があります。サーバー側では、1 つの代表的な形態としては、呼び出されるクラスをインスタンス化しておきます。そして、クライアントからリクエストがあると、そのインスタンスへの参照を渡し、そしてメソッド呼び出しをサーバーに存在するオブジェク

トに対して行えるようになるというようなイメージです。普通のオブジェクトは、new キーワードでインスタンス化しますが、RMI の場合はあらかじめサーバー側でインスタンス化されたオブジェクトを利用します。あるいはリクエストがあったときにサーバー側でインスタンスを作成して、作成したオブジェクトへの参照をクライアントに戻すというような動作も行います。こうした点が通常のメソッド呼び出しと違うだけで、あとは普通にメソッド呼び出しするのと同じ書式で記述できます。その意味ではネットワーク越しであることは、メソッド呼び出しの記述の点からは意識しなくてもいいというのが RMI のいちばんの特徴であると言えるでしょう。

スタブとスケルトン

模式的には、クライアントにあるオブジェクトから、サーバーにあるオブジェクトに存在するメソッドを呼び出し、引数を渡して返り値を得るという理解でかまいません。しかし、こうしたことを実現するために背後ではいろいろな仕事をしています。RMI のシステムではこうしたことがあまり表には出ないようにはなっていますが、いくつかの準備はどうしてもせざるを得なくなっています。

実際、メソッド呼び出しが行われる仕組みとして、「スタブ」と「スケルトン」というソフトウエアのモジュールが重要な役割を行います。RMI ではいずれも 1 つの class ファイルとして用意します。スタブはクライアント側、スケルトンはサーバー側で機能します。クライアント側でサーバー側のメソッド呼び出しが行われた時、実際にはまずスタブが呼び出されます。スタブはサーバー側のスケルトンと通信を行い、さらにスケルトンがターゲットのオブジェクトのメソッドを呼び出すという手順になります。返り値があるときには逆の順序を経て、クライアント側に値が戻されます。実際のメソッド呼び出しはこうしたいくらか複雑な手続きを経ます。また、スタブとスケルトンのやりとりも、ネットワーク処理など複雑な処理がからみます。いずれにしても、呼び出す側と呼び出されるオブジェクトの側の間にソフトウエアを介在させて、みかけは単なる呼び出しでも必要な処理が組み込まれるようになっているわけです。

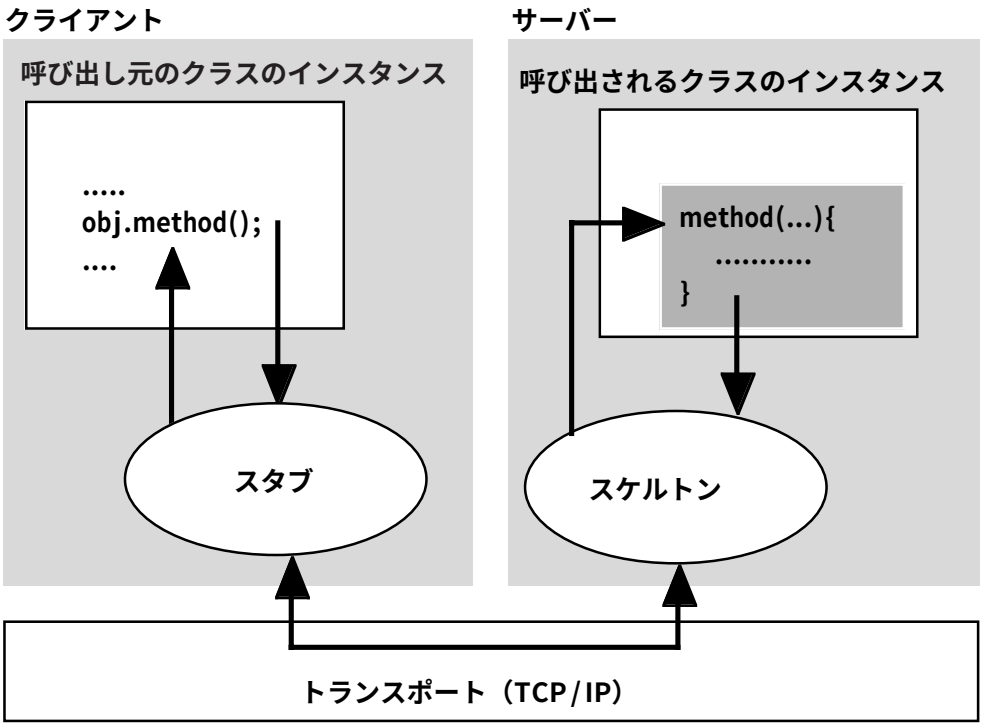

図 13-1　**スタブとスケルトンを経由してメソッド呼び出しが行われれる**

なお、スタブとスケルトンは、サーバーの呼び出されるクラスの class ファイルから自動的に生成されるので、プログラムを組んで作る必要はありません。自動生成するためのツールが、JDK に付属する rmic というもので、Mac OS の場合は、MRJ SDK に付属しています。

RMI の応用

RMI によって、ネットワーク越しのメソッド呼び出しが比較的簡単に実現します。これを利用すると、文字通りクライアント/サーバーアプリケーションが実現します。たとえば、サーバー側の呼び出されるクラスのメソッドで、データベースをアクセスしてデータの取り出しや書き込みができるようになっているとします。そのクラスを RMI を利用して呼び出すクライアントがあるというようなイメージを考えれば、3 階層システムを実現するということにもつながります。サーバー上のオブジェクトでビジネスロジックを実現するのです。ネットワーク処理として RMI を利用するということになります。

また、一連の処理を異なるコンピュータでこなすような分散オブジェクトを実現す

るベースにもなります。オブジェクト間の通信を行う機能を RMI を通じて実現するというわけです。

RMI は、Java をベースに開発するとなった場合、分散オブジェクトやクライアント/サーバーシステムを構築する極めてシンプルで強力な手段となります。プログラムの作成上は、お膳立てさえすれば、単にメソッドの呼び出しを行うだけで、ネットワーク越しのやりとりができてしまいます。そのお膳立ても、さまざまなネットワーク処理を行うことと比較すれば、非常にシンプルにできます。また、Java が実行できれば、プラットフォームに依存しないで利用できるのも大きなメリットです。たとえば、Mac OS をクライアントにし、Windows NT をサーバーにして RMI でのメソッド呼び出しができるわけです。

一方、Java の世界でしか利用できないというのはデメリットでもあります。他の言語で開発されたシステムとのやりとりとなると、RMI ではほとんど実現しないと言えるでしょう。また、サーバーが機能しなくなったようなときのバックアップ機能がシステムに組み込まれていないこともあるので、堅牢性を重視する場面では使いにくいという点も指摘されています。

13-2 RMI サーバーアプリケーションの作成

RMI サーバーは、現在の Java VM では、アプリケーションとして作成する必要があります。CodeWarrior Pro5 を使って、RMI サーバーをプログラミングし、rmic や JBindery といった MRJ SDK で提供されているツールを使って最終的に仕上げる手順も含めて解説を行いましょう。

プロジェクトの用意

RMI を使ったシステムを構築する場合、クライアントとサーバーの 2 つのアプリケーションあるいはアプレットを作ることになるでしょう。その場合でも、開発に必要な資源を 1 つのプロジェクトにまとめておくというのが 1 つのやり方だと思われます。そこで、この章のサンプルでは、CodeWarrior Pro5 の RAD ツールを使ってクライアントのプログラムを作成し、同じプロジェクトにサーバーのプログラム作成ターゲットを追加するという方針で臨むことにします。以下、クライアント向けにアプリケーションウィザードを使って AWT を使ったフレームを 1 つ持ったアプリケーション（名前は TrialRMI）を作ったプロジェクトで作業するものとします。

作成したプロジェクトには、System Classes というグループが最初からあり、そこに JDKClasses.zip と swingall.jar が含まれています。今回のサンプルでは、swingall.jar はなくてもかまいません。そして、Application1.java と Frame1.java の 2 つのソースファイルがあります。これからいくつかソースを増やすのでややこしくならないように、Server Sources と Client Sources というグループを作り、Application1.java と Frame1.java の 2 つのソースファイルは Client Sources グループにドラッグして移動しておきます。

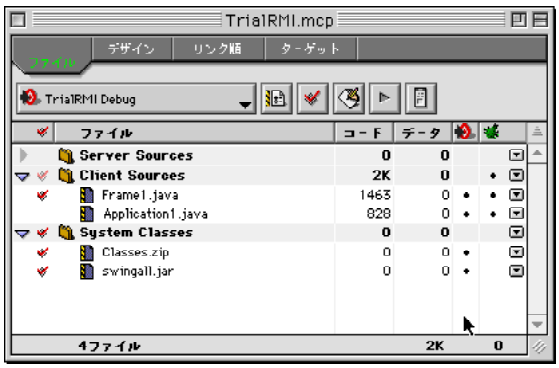

図 13-2　ソースファイルを分類できるようにしておく

グループを作成するには、プロジェクトのウインドウの上部で「ファイル」が選択されいている状態で、「プロジェクト」メニューの「新規グループの作成」を選択します。

次に、このプロジェクトに、サーバーアプリケーション生成用のターゲットを作成します。プロジェクトのウインドウの上部で「ターゲット」を選択している状態で、「プロジェクト」メニューから「新規ターゲットの作成」を選択します。次のようなダイアログボックスが表示されるので、ここではターゲット名を Server Application としました。

図 13-3　新しくターゲットを作成する

♦ターゲットの設定を変更する

新しくターゲットを作成したら、そのターゲットの設定を変更します。以下ポイントを説明していくことにします。ターゲットの設定は、たとえばプロジェクトのウインドウでターゲットの項目をダブルクリックするなどして表示されるダイアログボックスで設定します。

まず、「ターゲット設定」ですが、リンカとして「Java Linker」を選択します。CodeWarrior Pro5 からは、Mac OS 向けのアプリケーションを生成するポストリンカはリンカに組み込まれたので、ポストリンカを選択する必要はありません。

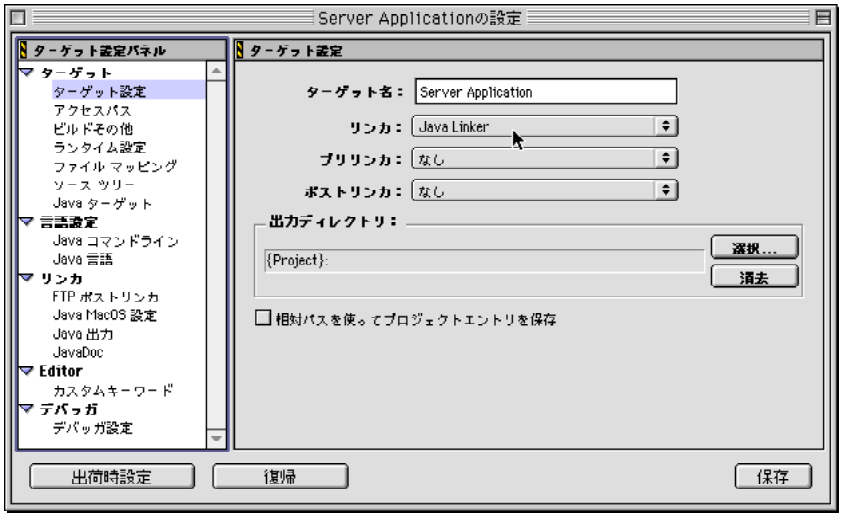

図 13-4　Java Linker **を選択する**

次に「Java ターゲット」を選択します。まず、ターゲットタイプは「アプリケーション」を選択しておきます。さらにここでは、実際に main メソッドのあるクラス名を指定しなければなりません。この段階ではまだソースコードを紹介していませんが、後から説明するように、TestServerApp という名前のクラスでアプリケーションクラスを定義するので、ここでその名前を入れておくことにします。

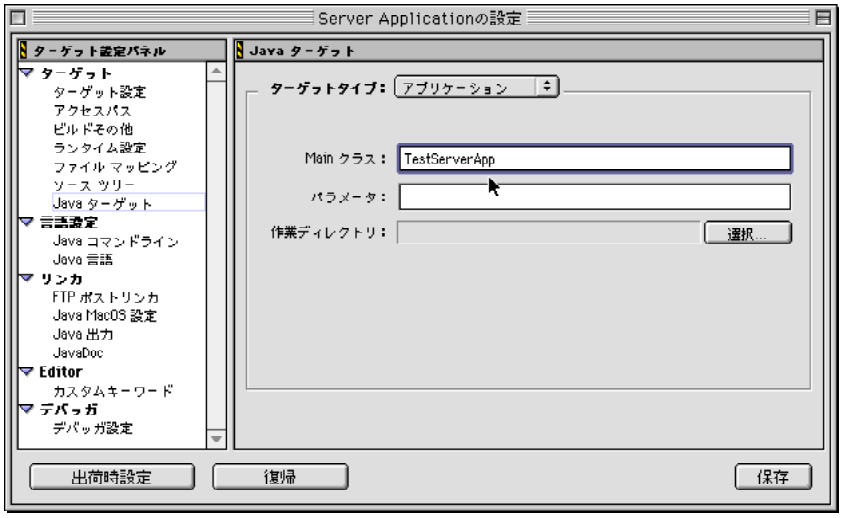

図 13-5　**アプリケーションとして実行する**

次に「Java MacOS 設定」で、作成されるアプリケーション名をここでは「TestServer」としました。既定の名前でもかまわないのですが、今回のサンプルでも 3 つのアプリ

ケーションを作り、うち 2 つは CodeWarriror が生成しますので、同じ名前のファイル名を付けていて上書きされるとよくありません。そこで、アプリケーションにはそれぞれ異なる名前を付けておくことにします。

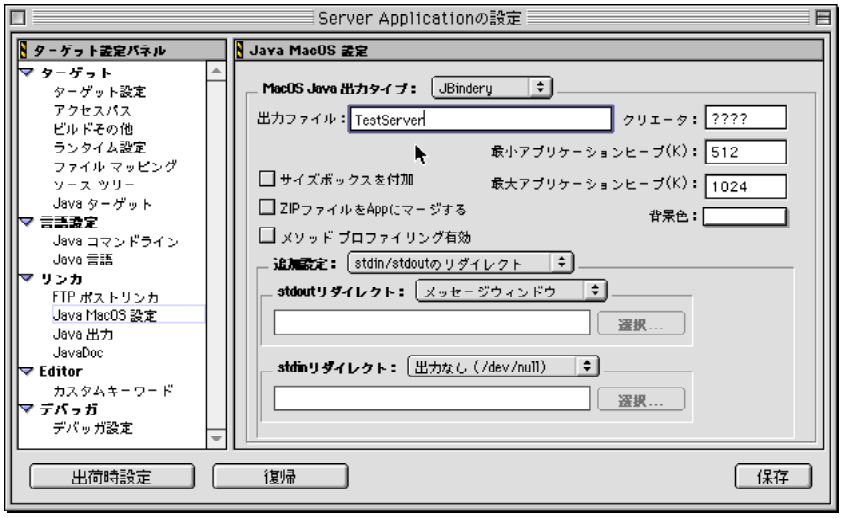

図 13-6　生成されるアプリケーションのファイル名を指定する

さらに「Java 出力」では、出力タイプとして「クラスフォルダ」を選択しておき、生成したクラスをフォルダにため込む形式にします。そして、この図のように、Server Classes というフォルダにコンパイル結果がため込まれるようにします。

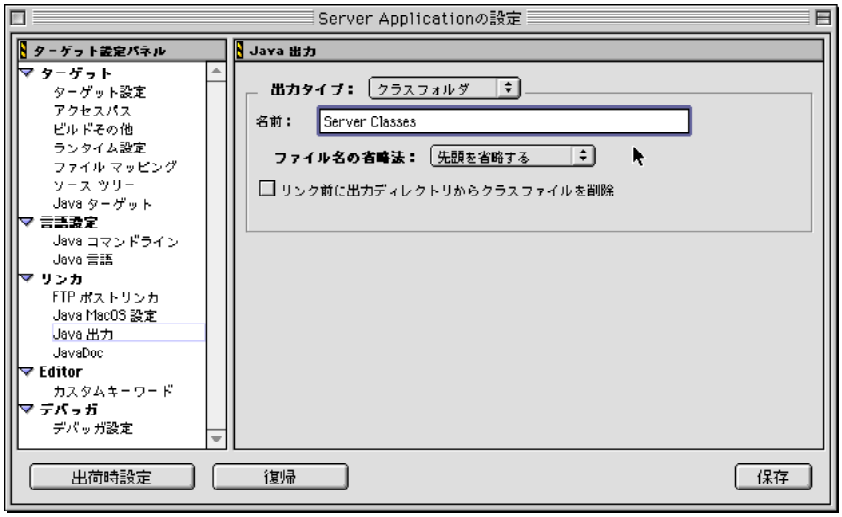

図 13-7　コンパイル結果はクラスごとのファイルで得られるようにする

通常は、ファイル名の長さの問題もあるので、JAR ファイルにコンパイル結果をア

ーカイブします。しかしながら、RMI では、スタブとスケルトンのクラスを作りますが、それらのクラスが独立した class ファイルで得られます。作成するアプリケーション全体的に JAR は使わない方針ですべてクラスごとの class ファイルを作って動作させることにします。もちろん、完全に作り込んだ後に、JAR ファイルにまとめるのはいいのですが、RMI を使ったシステムを作る段階ではクラスごとにファイルが得られるようにしておくのが何かと取り扱いやすいでしょう。

さらに、RMI サーバー側でも、Java の基本クラスライブラリである Classes.zip を必要とします。プロジェクトのウインドウの上部で「ファイル」を選択し、ファイル一覧で、Classes.zip を選択します。そして、「ウィンドウ」メニューから「プロジェクトインスペクタ」を選択して、Classes.zip に対するインスペクタを表示します。そして、インスペクタウィンドウの上部で「ターゲット」を選択し、Server Application のチェックボックスをオンにして、「保存」ボタンをクリックします。これで、Classes.zip は、Server Application のターゲットでも利用できるようになります。

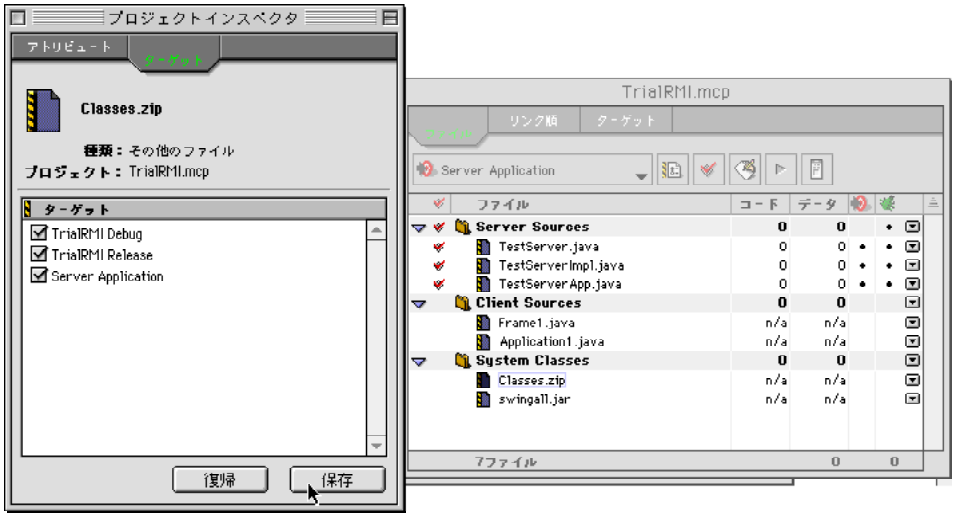

図 13-8　Classes.zip をサーバーでも利用するように設定しておく

インタフェースの定義

RMI を利用するサーバー側にはもちろんクライアントから呼び出されるメソッドが存在します。そのメソッドはもちろん、クラスとして定義されるのですが、クラスとしての定義は、クライアントでも必要ということになります。

サーバーのクラスをいきなり定義するのではなく、呼び出されるクラスやメソッドをインタフェースとして定義し、そのインタフェースを実装するクラスをサーバーで

機能させるという段階を踏みます。

まず、インタフェースクラスですが、java.rmi.Remote インタフェースを拡張して定義する必要があります。これは、java.lang.Object が通常のオブジェクトの基本形であるのに対して、Remote はリモートオブジェクト、つまりネットワークを介しても存在しうるオブジェクトの基本形であるということになります。また、例外として java.rmi.RemoteException を投げるように定義する必要もあります。

たとえば、ここでは、まず TestServer というインタフェースを次のように定義したとします。ファイルは、TestServer.java という名前で保存しますが、プロジェクトでは、Server Sources グループに所属させておくと整理されるでしょう。

```
import java.rmi.*;

//RMI で呼び出されるクラスのインタフェース定義
public interface TestServer extends Remote
{
    //呼び出されるメソッドを定義する
    public String getInfo(String str)
        throws RemoteException;
}
```

プログラム自体はシンプルで、作成したクラスでは、1 つのメソッド getInfo だけが定義されています。まず、inteface でクラスを定義するときに、Remote を拡張するのが 1 つのポイントです。そして、RMI によって呼び出されるメソッドは、RemoteException が投げられるという定義を含めておくということです。後は、クライアントから呼び出される必要なメソッドをいくつでも定義しておけばかまいません。

このソース TestServer.java は、サーバーアプリケーションのターゲットである Server Application に登録するのはもちろんですが、クライアントのアプリケーションのターゲットでも読み込まれるようにしておく必要があります。たとえばここでは Java アプリケーションウィザードを使って作成したプロジェクトだと、TrialRMI Debug、TrialRMI Release のターゲットにも所属させておく必要があります。そうしないと、クライアント側のアプリケーションのコンパイルが通りません。クライアント側でも、サーバーにある TestServer を呼び出すわけですから、TestServer の定義自体は手元にもないといけないと解釈すればいいでしょう。

実装クラスの作成

インタフェースとして定義したサーバー側のクラスの実装を行います。ここでの例だと、TestServer というインタフェースをインプリメントしたクラスであるという必

要がありますが、サーバーとしての機能を組み込むために、java.rmi.server.UnicastRemoteObject というクラスを拡張して定義します。UnicastRemoteObject は異なるコンピュータに存在するオブジェクトを実現する機能を持ったクラスで、そのクラスを拡張することで、目的とするサーバークラスが作れるということです。

では、TestServer を実装する TestServerImpl クラスを定義します。ファイル名は、TestServerImpl.java としました。ソースはプロジェクトでは Server Sources に分類しておけばよいでしょう。所属させるターゲットとしてはサーバー（ここでは Server Application）だけでかまいません。次のようなプログラムを作成しました。

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.*;

//サーバーを実現するクラス
public class TestServerImpl extends UnicastRemoteObject
    implements TestServer
{
    private     int    callCount = 0;
    //この形式のコンストラクタが必要になる
    public TestServerImpl() throws RemoteException
    {
        super();    //スーパークラスのコンストラクタを呼び出す

        System.out.println(
            (new Date(System.currentTimeMillis()).toString())
            + ":Test Server Starting");
                //コンソールに現在時刻とメッセージを表示
    }

    //メソッドの本体を記述する
    public String getInfo(String str)    {

        callCount++;     //呼び出し回数のカウンタを 1 つ増加する
        System.out.println(
            (new Date(System.currentTimeMillis()).toString())
            + ":Get:" + str + "/Call Count = " + callCount);
                //コンソールに現在時刻と、引数で受け取った文字列を表示

        return("文字列の長さは、"+str.length()+"文字です。");
            //文字列の長さを含む文字列を戻す
    }
}
```

まず、TestServer をインプリメントしているということで、getInfo メソッドの本体をここで必ず定義しなければなりません。言い換えれば、TestServer 側ではプログラ

ム本体は定義できないので、ここで記述するわけです。getInfo 自体は、引数で得られた文字列の文字数をカウントして、その結果を文字列として返すという単純なものです。このとき、メソッドが呼び出されたことが目で見て分かるように、Java コンソールにテキストで引数や現在の日付と時刻などを println で出力しています。

そして、この TestServerImpl クラスのコンストラクタを定義するのですが、ここで、RemoteException を投げるように定義を加えておく必要があります。このコンストラクタでは、実質的には super しか呼び出してはいませんが、それだけのコンストラクタでも記述は必要です。コンストラクタを省略すると、RemoteException を投げずに引数のないコンストラクタがあるように解釈するのですが、UnicastRemoteObject の引数なしコンストラクタは RemoteException を投げるように定義されています。したがって、同じ形式でサーバーの実装クラスでも定義が必要になると考えればよいでしょう。

以上の要件が満たされていればそれだけでサーバーになります。ネットワーク処理などのややこしいことは一切プログラミングする必要がなく、それらは、たとえば UnicastRemoteObject などの背後で定義されていると考えてプログラムを作ればよいでしょう。

起動アプリケーションの作成

RMI サーバーは、現在のバージョンでは必ずアプリケーションとして作成する必要があります。クライアントはアプリケーションでもアプレットでもかまいません。呼び出しを受け付けるクラスはアプリケーションとして起動されている必要があります。アプリケーションを作る方法としては、独立したクラスを作るか、あるいは前の TestServerImpl に main メソッドを定義するかというのが代表的な方法でしょう。ここでは前者の方法を取ることにします。

以下のようなクラスを定義して、TestServerApp.java というファイルに保存しました。プロジェクトでは、Server Sources のグループに分類しておきますが、このソースも、もちろん、サーバーアプリケーションのターゲットにだけ所属させておくだけでかまいません。

```
import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.*;

//サーバーとして機能するアプリケーション
public class TestServerApp
{
```

```
    public static void main(String args[])     {
        try    {
            System.setSecurityManager(new RMISecurityManager());
                //セキュリティの管理を加える
            TestServerImpl s = new TestServerImpl();
                //呼び出されるサーバー側クラスのインスタンスを生成する
            Naming.rebind("TESTSERVER", s);
                //インスタンスに結び付ける名前を登録する
        }
        catch(Exception e)     {
            System.out.println(e.getMessage());
        }
    }
}
```

サーバーアプリケーションの最初のポイントは、サーバーのクラスのインスタンスを生成するところにあります。ここで定義している TestServerImpl というクラスは、もちろん、インスタンス化しないと利用できないことは言うまでもありません。そのインスタンス化はサーバーのアプリケーションで起動時に行うのが 1 つの代表的な方法です。

インスタンス化されたオブジェクトが存在することが、コンピュータの外部から参照できなければいけません。そのために、rmiregistry というメカニズムを利用します。具体的なことは後から説明しますが、ここでは、rmiregistry というサーバーがあって、RMI 呼び出し可能なメソッドを持つオブジェクトのディレクトリを持っており、クライアントからの要求で必要な情報を与えると考えて下さい。

TestServerImpl のインスタンスを作ると、それを Naming クラスのクラスメソッドである rebind メソッドを利用して、サーバーに登録します。そのとき、ここでは TESTSERVER という名前でそのオブジェクトのインスタンスを参照できるように名前を設定しました。rebind の引数はあえて TestServer とはせずに別の名前にしました。後でクライアントでどこに指定する文字列と対応しているかをよくチェックしてください。

サーバーアプリケーションではセキュリティ関連の設定を行っておきます。一般には、RMISecurityManager で与えられる機能を追加することになります。System クラスのクラスメソッドである setSecurityManager を使います。ここでのセキュリティの設定によって、リモートでやりとりされたオブジェクトなどの影響によって、サーバーにダメージを与えることをなるべく防いでいると考えればよいでしょう。

♦アプリケーションをメイクする

ここまでが CodeWarrior での作業になります。ソースファイルを準備して設定など

も行ったら、たとえば Command+M などでメイクしておきます。正しくコンパイルされると、ここでは、TestServer というアプリケーションが作成されているはずです。また、Server Classes というフォルダには、ソースファイルをコンパイルした結果の class ファイルが出来上がっているはずです。

図 13-9　サーバーアプリケーションをコンパイルした結果

スタブとスケルトンの作成

次にスタブとスケルトンを用意します。そのために、rmic というツールを利用しますが、MRJ SDK の中の Tools:JDK Tools フォルダにあります。実際にサーバーとして機能しているクラスは、ここまでの例では、TestServerImpl です。そのコンパイル結果は、TestServerImpl.class として得られているはずです。rmic を使えば、コンパイル結果の TestServerImpl.class からスタブとスケルトンが生成されます。スタブは、TestServerImpl_Stub.class、スケルトンは TestServerImpl_Skel.class という名前で生成されます。つまり、元のクラスファイルのファイル名部分に、_Stub あるいは_Skel という文字列がつなげられたファイル名で自動的に生成されるのです。

スタブとスケルトンの作成は次のように作業します。まず、ファイル TestServerImpl.class を rmic にドラッグ&ドロップして、rmic を起動します。この作業ができるように、コンパイル結果をクラスファイルとして得られるように設定したというわけです。

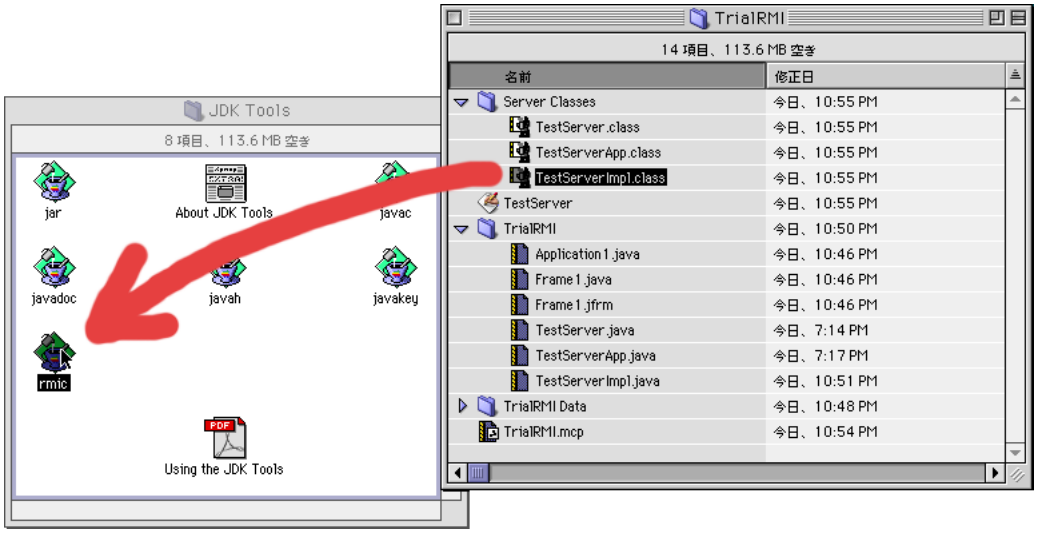

図 13-10　サーバーの実装クラスのコンパイル結果を rmic にドラッグ&ドロップ

こうして rmic を起動すると、最初の画面は次のようになっているはずです。Classnames には、ドラッグしたファイルのクラス名だけの「TestServerImpl」が設定されています。ここは修正する必要がありません。クラスパスはちょっと見づらいですが、MRJLibraries フォルダにあるおなじみの MRJ のクラスファイルを参照するようになっています。

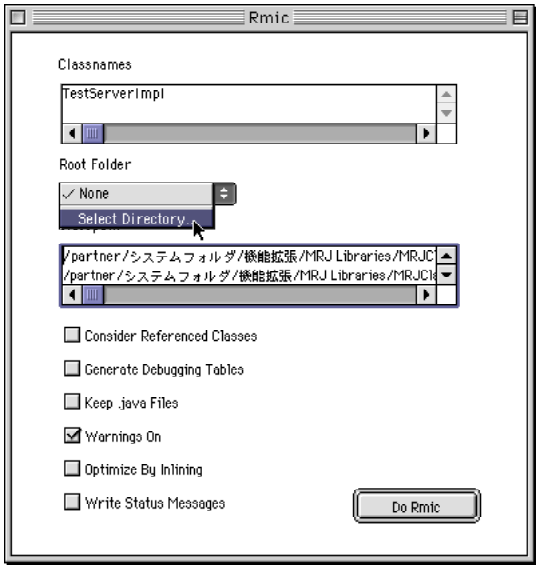

図 13-11　rmic のダイアログボックス

ここで、図にあるように、Root Folder のポップアップメニューから Select Directroy を選択します。ここはルートフォルダというように、クラスファイルの位置を指定し

ます。ここでは、表示されるダイアログボックスで、プロジェクトと同じフォルダに作成された Server Classes フォルダを参照しておくようにします。

このルートフォルダがドラッグ&ドロップされてきたファイルに応じて自動的に設定されないのには理由があります。たとえば、あるクラスが特定のパッケージに含まれるものの場合、パッケージの階層構造に合わせてフォルダが作られ、その中にクラスファイルが入っています。そのようなクラスを参照する場合、クラスのあるフォルダではなく、パッケージ階層のルートフォルダをここで指定する必要があるからです。

ここで、クラスパスの指定も必要なのですが、ちょっと手抜きながら手軽に確実に設定できる方法を説明しましょう。Root Folder を設定して、Do Rmic ボタンをクリックして動作させると、コンソールのウインドウにエラー表示されるかと思います。

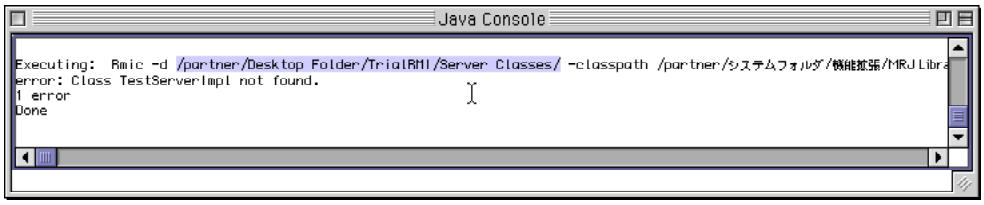

図 13-12　そのまま実行するとエラーが出る

このとき、rmic コマンドの-d オプションの引数（ここでは、/partner/Desktop Folder/TrialRMI/Server Classes/）の部分を選択してコピーし、それを rmic のダイアログボックスの Classpath の中に加えます。

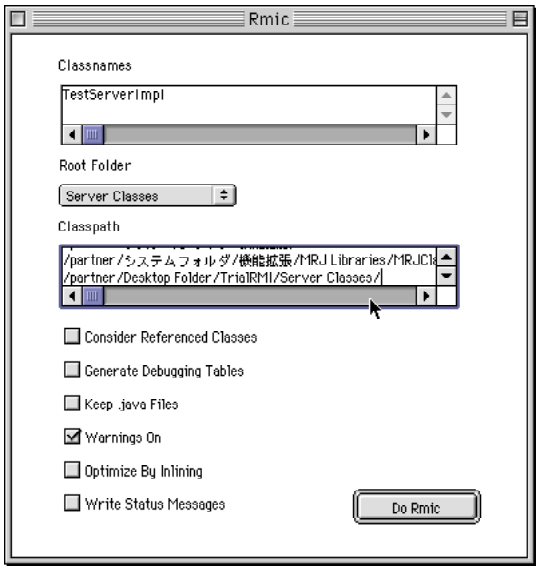

図 13-13　クラスパスにサーバーのクラスの存在する場所を指定した

つまり、TestServerImpl.class の存在するパスをクラスパスに指定する必要があるのです。Classpath のテキストボックスは作業しづらいですが、新しい行を加えて、コンソールに表示された内容をコピー&ペーストすれば、少しは作業は手軽になると思われます。

こうして、Do Rmic ボタンをクリックすると、スタブとスケルトンが作成されます。これらの class ファイルは、ここでは、TestServerImpl.class と同じフォルダに作成されます。ここで、Keep .java Files のチェックボックスを入れておけば、スタブやスケルトンの class ファイルのソースファイルが残るので、スタブやスケルトンがどんなメカニズムなのかを知りたい場合にはソースを作っておくと良いでしょう。ですが、そうしたことは知らずに、スタブやスケルトンと言うことを概念的に知っておくだけで RMI を利用したシステムは十分に作ることができます。

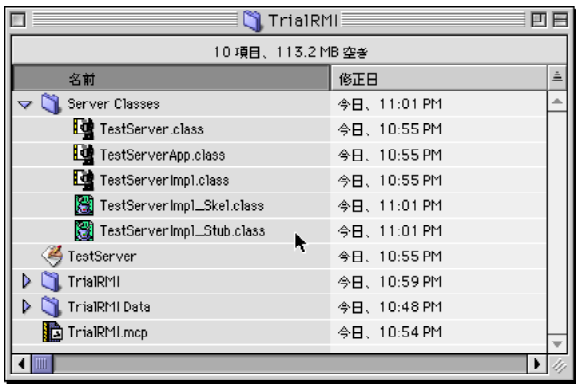

図 13-14　スタブとスケルトンが作成された

以上で、RMI サーバーとして機能するアプリケーションは完全に作成されました。これまでのような手順で作成した TestServer というアプリケーションは、クラスパスに Server Classes を含むので、アプリケーションは、スケルトンのクラスを自動的に認識します。つまり、スケルトンの呼び出しがあれば、アプリケーションからはアクセス可能になっているということです。

ここで、スタブはクライアントで使うものということであれば、サーバー側には必要ないと思うかも知れませんが、実際に稼動させたところ、サーバー側ではスケルトンだけでなく、スタブも必要になるようです。スタブのクラスファイルがない場合には、そのクラスが見つからないという例外が出てしまいます。いくつか調べた書籍の中では、「サーバー側はスケルトンだけでいい」と明記されているものもあったのですが、このことが Mac OS 独自の事情なのかは分かりませんでした。しかしながら、

いずれにしても、ここで紹介したサーバーを実行する場合、スタブも必要です。具体的には、次の rmiregistry がスタブを必要とするようです。Naming.rebind メソッドの呼び出し時に、スタブも参照しているようです。

rmiregistry を用意する

サーバー側でインスタンス化した RMI 対応のオブジェクトのディレクトリのような機能を持つ rmiregistry という機能が必要になります。つまり、RMI で呼び出されるサーバーとは別のサーバーが必要であると考えれば良いでしょう。クライアントで指定した名前を持つ公開されたオブジェクトのインスタンスを戻すような機能を持っているサーバーと考えればいいわけです。「登録サーバー」という言い方がなされることがありますが、的を得た表現だと思います。

この rmiregistry を機能させる 1 つの方法は、rmiregistry サーバーアプリケーションを起動することです。もう 1 つの方法は、RMI に対応したサーバーアプリケーションで、rmiregistry の機能を呼び出すという方法です。

まず 1 つ目の方法ですが、一般の JDK では、rmiregistry というアプリケーションが付属していることから、そのアプリケーションを起動することで、rmiregistry の機能が働き、RMI に対応したサーバーアプリケーションの利用ができるようになります。MRJ SDK にはそのようなアプリケーションはありませんが rmiregistry のアプリケーション機能は、基本クラスの中に完全にプログラムが組み込まれており、JDK の rmiregistry も、単にそのクラスを実行しているに過ぎないのです。

そこで、Mac OS で動く rmiregistry を MRJ SDK に付属する JBindery で作成することにします。ただし、ここで、実際に RMI として公開するクラスのスタブがそのアプリケーションから参照できるようになっていないと機能しなかったので、実質的には汎用アプリケーションとしては利用できないことになります。つまり、ここでのサンプル専用ということになるのですが、ともかく動かす方法はこれしかないようです。

まず、MRJ SDK にある JBindery を起動します。まず、Class name として、sun.rmi.registry.RegistryImpl をキータイプします。これは、JDKClasses.zip に組み込まれた rmiregistry の実装クラスで、アプリケーションとして起動可能なクラスでもあるのです。

図 13-15　JBindery を起動し、起動するクラスをキータイプする

次に左側の Classpath のアイコンをクリックして、クラスパスの追加を行います。ダイアログボックスで、Add Folder ボタンをクリックし、プロジェクトと同じフォルダにある Server Classes フォルダを指定します。つまり、スタブの存在するフォルダをここで指定しておくのです。

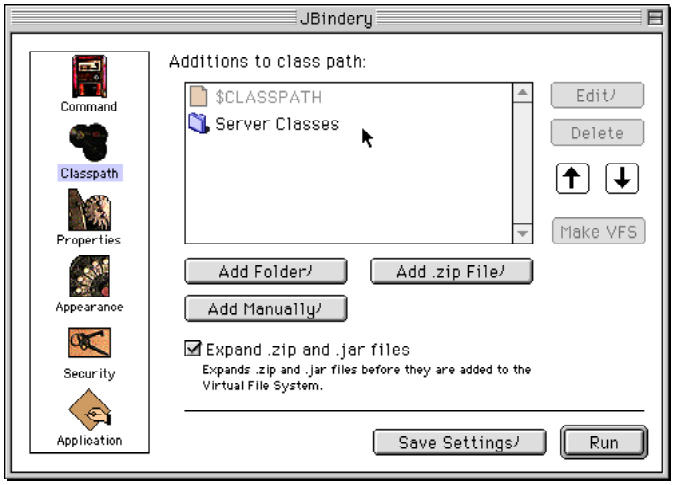

図 13-16　スタブの存在するフォルダをクラスパスに加えておく

そして、Save Setting ボタンをクリックして、アプリケーションを作成します。ここでは、プロジェクトと同じフォルダに、Mac rmiregistry としてファイルを保存しました。結果的に RMI のサーバーアプリケーションである TestServerApp と同じフォルダに作成されることになります。

♦RMI サーバーから rmiregistry を呼び出す方法

rmiregistry を実現するもう 1 つの方法は、サーバーアプリケーションで、rmiregistry を生成してしまうことです。これだと、別途アプリケーションを起動しなくても、RMI のサーバーアプリケーションを起動するだけでかまいません。その場合、サーバーアプリケーションに 1 つのステートメントを加えるだけです。ここまでの例だと、TestServerApp.java を次のように作れば、RMI サーバーアプリケーションで rmiregistry を呼び出すことができます。

```
import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.*;

//サーバーとして機能するアプリケーション
public class TestServerApp
{
    public static void main(String args[])    {
        try    {
            LocateRegistry.createRegistry(1099);
                //rmiRegistry を自分で起動する場合
            System.setSecurityManager(new RMISecurityManager());
                //セキュリティの管理を加える
            TestServerImpl s = new TestServerImpl();
                //呼び出されるサーバー側クラスのインスタンスを生成する
            Naming.rebind("TESTSERVER", s);
                //インスタンスに結び付ける名前を登録する
        }
        catch(Exception e)    {
            System.out.println(e.getMessage());
        }
    }
}
```

前に紹介した TestServerApp.java との違いは、LocateRegistry クラスのクラスメソッド createRegistry を呼び出して、rmiregistry の機能をアプリケーション内から呼び出して起動するようになっているというところです。1099 は RMI の通信で利用するポート番号です。

この場合でも、スケルトンだけでなくスタブのクラスファイルは必要です。スケルトンは、この場合 Server Classes フォルダにあるので、単にスタブも作られたまま動かさないで、そのまま Server Classes フォルダに入れておけばいいということになります。

13-3 RMI のクライアントの作成

前節で説明した RMI サーバーのメソッドを呼び出すクライアントアプリケーションを作成してみました。ユーザーインタフェースを持たせるのですが、CodeWarrior Pro5 からの搭載された RAD 機能を使って手軽に仕上げるということをやってみました。

ユーザーインタフェースの作成

最初にも説明したように、CodeWarrior Pro5 の Java アプリケーションウィザードを使って、プロジェクトを用意しました。AWT をベースにしており、アプリケーションは Application1.java、ウインドウは Frame1.java というソースで最初に自動生成されますが、それをそのまま使うことにします。また、プロジェクト名は TrialRMI としましたので、プロジェクトのフォルダやファイル、あるいは、ソースコードを入れるフォルダの名前にこの名前が使われます。また、Java アプリケーションウィザードが生成した Application1.java と Frame1.java は、TrialRMI というパッケージに含まれることになるので、正確にはクラスは、TrialRMI.Application1 と TrialRMI.Frame1 ということになります。

そして、Frame1 に対して、以下のようにコンポーネントを配置しました。Label が 3 つ、TextField が 3 つ、そして Button が 1 つとなっています。ボタンは、button1 とう name プロパティになっています。TextField は、Server に対応するものが textField1、引数に対応するものが textField2、返り値に対応するのが textField3 という name プロパティになっています。

プログラムは、指定したサーバーにある RMI 対応クラスのメソッドを呼び出しますが、引数は TextField で指定したものを用います。そして、呼び出したメソッドの戻り値を、やはり TextField に設定するというような動作を行うものとします。

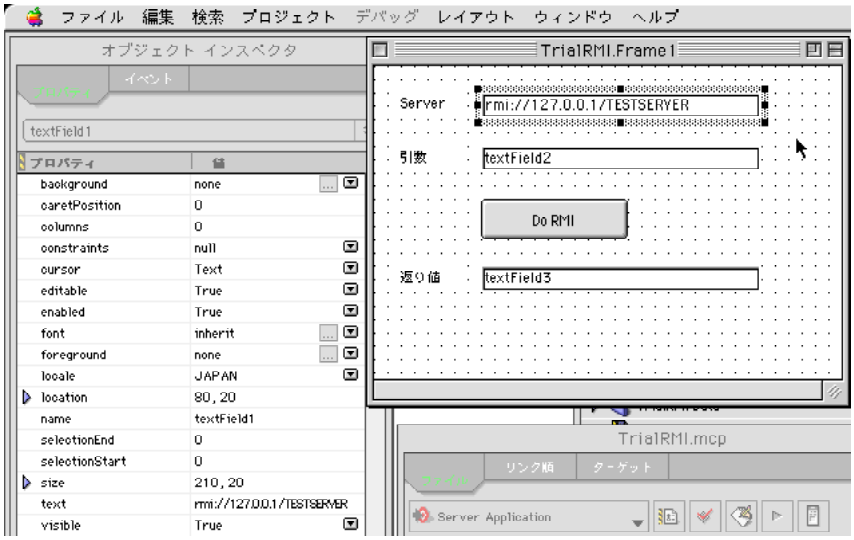

図 13-17　RAD ツールでユーザーインタフェースを作成する

RMI によるリモートメソッドの呼び出し

ユーザーインタフェース部分が作成できれば、ボタンをクリックして、サーバーにあるメソッドを呼び出す部分を作り込むことにします。まず、Button をクリックすると言うことは、Action イベントが発生します。そのイベントに対応するメソッドを Frame1 に付け加えるのですが、オブジェクトインスペクタの上部で「イベント」を選択します。そして、その下にあるポップアップメニューで「button1」を選択します。そして、actionPerformed の項目の右側の空欄をクリックすると、作成されるメソッド名が自動的に現れます。そして、return キーを押すと、ソースのウインドウが開き、actionPerformed メソッドの部分が表示されるので、そこにプログラムを追加します。

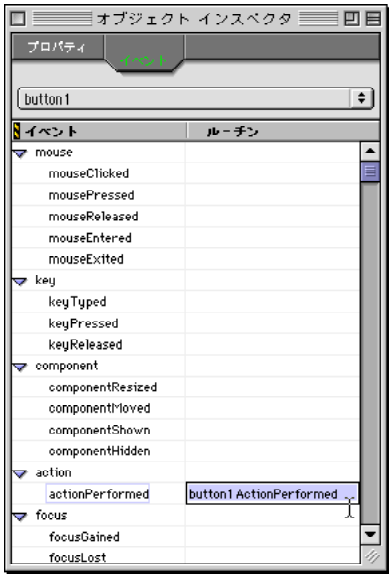

図 13-18　ボタンのアクションイベントのメソッドを追加する

♦リモートメソッドの呼び出し処理

最初は、button1ActionPerformed メソッドの定義だけで、中身のプログラムはありません。そこで、以下のように、メソッドの中身を完成させることにします。import 文はもちろん、ソースファイルの冒頭に記述します。

```
import java.rmi.*;                          //******この行を追加
import TestServer;                          //******この行を追加

    :

    //ボタンをクリックした時のメソッド
    public void button1ActionPerformed(java.awt.event.ActionEvent e)
    {
        try    {
            TestServer rs = (TestServer)Naming.lookup(textField1.getText());
                //textField3 の内容から、指定したクラスを探し、
                //      そのクラスのインスタンスへの参照を得る
            String r = rs.getInfo(textField2.getText());
                //サーバーにあるオブジェクトの getInfo メソッドを呼び出した。
                //      引数は textField1 の値
            textField3.setText(r);
                //返り値を textField2 に設定した
        }
        catch(Exception ex)    {
            System.out.println(ex.getMessage());
        }
    }
```

ポイントを説明しましょう。まず、RMI サーバー側で呼び出し可能になっている

オブジェクトのインスタンスを得たいのですが、そのためのメソッドとして、Naming クラスにあるクラスメソッドの lookup メソッドを利用します。この引数に、サーバーを指定します。すると、指定したサーバーの rmiregistry を参照して、指定した名前で公開されているサーバークラスのインスタンスへの参照を返り値として返します。それをキャストして、得たい型のオブジェクト参照とします。

lookup の引数には、「rmi://アドレス/名前」という形式の URL で指定します。まず、プロトコルは rmi ということです。アドレスの部分はドメイン名やあるいは IP アドレスを記述します。rmiregistry とともに起動しているサーバーアプリケーションの存在するコンピュータを指定することになります。一般には自分自身を参照する場合は localhost を指定できるのですが、Mac OS ではこの指定が効かないので、自分自身を指定する手軽な方法としては、IP アドレスとして 127.0.0.1 を指定するのが無難でしょう。そして、URL の名前の部分は、Naming.rebind で登録した時の名前を指定します。前の節で説明したサーバーアプリケーションでは「TESTSERVER」という名前で rmiregistry に登録したので、呼び出す側でも同じ名前にしておく必要があります。たとえば、同じマシンでサーバーとクライアントを動かした場合には、「rmi://127.0.0.1/TESTSERVER」を、Naming.lookup の引数に指定すれば、サーバーのオブジェクトへの参照が得られるというわけです。

こうして別のコンピュータにあるオブジェクトへの参照が得られれば、そのオブジェクトでサポートしているメソッドを、普通のオブジェクトと全く同じように利用できます。ここでは、getInfo メソッドが定義されているので、リモートのオブジェクトを参照している変数 rs に対して、getInfo メソッドを適用できます。

生成しているオブジェクトを参照したりする場面では RMI 独特ではありますが、その後は普通にメソッドを利用できます。もちろん、メソッドの呼び出しは、実際にスタブを呼び出すことになり、スタブが通信をしてサーバーに引数を送ります。そして、サーバー側のオブジェクトで記述された処理を実行して結果を戻すということをやるわけです。

セキュリティの設定

クライアント側のセキュリティ設定を行う必要があります。RMI のメカニズムが働くと、サーバーからクライアントに対してクラスが転送されることになります。ここで、正しく機能しないクラスを排除するために、セキュリティの設定を加える必要があります。プログラム的にはサーバーと同一で 1 つのメソッド呼び出しを加えるだ

けです。ここでのサンプルでは、Frame1 のコンストラクタで以下のように呼び出すことにします。

```
public Frame1()
{
        System.setSecurityManager(new RMISecurityManager());
                //RMI 呼び出しができるようにセキュリティの管理を行う
}
```

なお、アプリケーションで実行する時には、このセキュリティの設定がなくても実行はできるようです。しかしながら、安全性などを考えれば、基本的にはセキュリティマネージャの設定は必要だと思われます。

クライアントアプリケーションの構築

クライアント側のアプリケーション作成に必要な設定を行っておきましょう。ここでは、Java アプリケーションウィザードで生成したターゲットからの変更内容だけを紹介します。

まず、サーバーアプリケーションと同じように、Java 出力の設定で、出力タイプを「クラスフォルダ」に指定します。出力するフォルダはここでは Client Classes にしておきます。これも、やはりスタブのクラスファイルをコピーして済ませるようにするための措置だと考えて下さい。

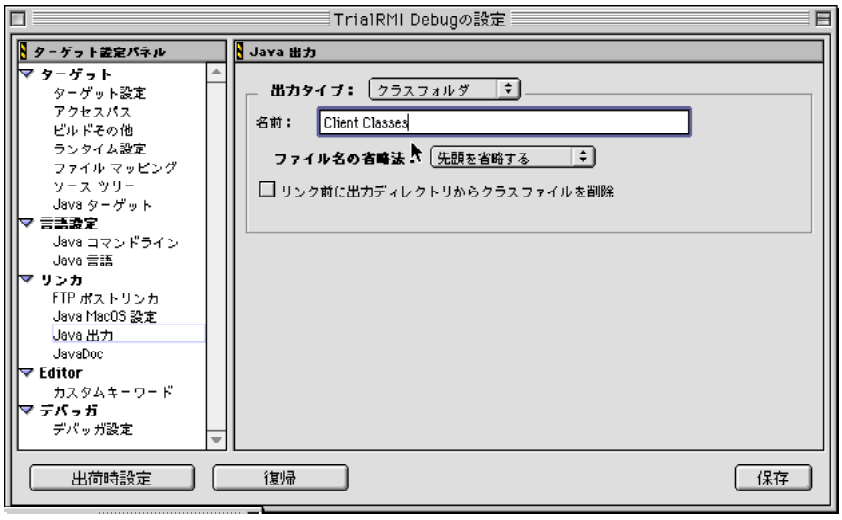

図 13-19　コンパイル結果はクラスファイルで出力する

そして、Java MacOS 設定では、作成するアプリケーション名を指定します。ここ

では、TrialRMI Client という名前にしました。

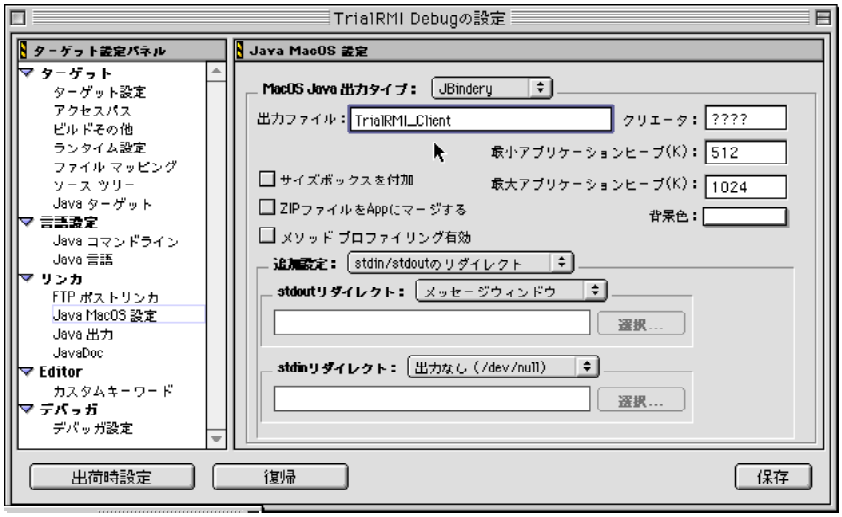

図 13-20　アプリケーション名を設定する

こうして、Command+M などでメイクすると、アプリケーションが生成されます。アプリケーションの TrialRMI Client がプロジェクトと同じフォルダに作成され、Client Classes フォルダが作成され、そこにコンパイルされたクラスファイルが作られます。Application1 と Frame1 は TrialRMI パッケージ内なので、TrialRMI フォルダがさらに中に作られます。こうして、作られた Client Classes フォルダに、rmic で作成したスタブのクラスファイルである TestServer_Stub.class をコピーしておきます。これまでの手順で作っていれば、Server Classes フォルダにあるはずなので、Finder 上でコピーをしておけば良いでしょう。

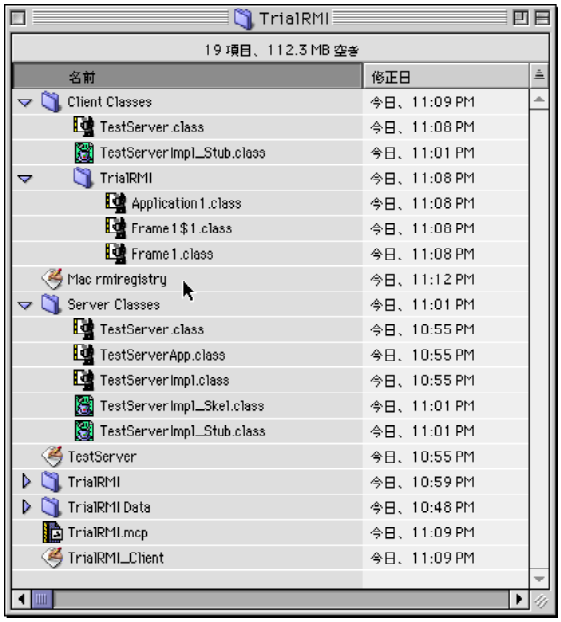

図 13-21　生成したクライアントのクラスフォルダにスタブをコピーしておく

13-4 サンプルプログラム

サンプルプログラムの動作例を説明し、まだ紹介していないソースについても掲載しておきます。この章のこれまでの部分で紹介したソースはここでは紹介していませんので、前の部分を御覧ください。

一連のアプリケーションの実行例

これまでに見てきたように、アプリケーションは 3 つ作られています。これらのアプリケーションは、一部に起動する順序を正しく行わないといけないものもありますので、動作例として起動方法などを説明しましょう。

まず、サーバーから起動しますが、最初に、Mac rmiregistry をダブルクリックして起動します。少し待って完全に起動するのを待ちます。ただし、このアプリケーションはメッセージなどは一切出しませんので、起動したかどうかは、アプリケーションメニューなどを参照してください。

その後に、TestServer アプリケーションを起動します。正しく起動できれば、プログラムに記述したように、起動を示すメッセージが Java コンソールに表示されます。

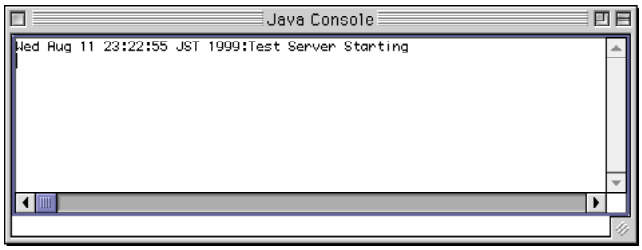

図 13-22　**起動した** TestServer

これで、RMI サーバーが起動して、インスタンスが生成され、rmiregistry に登録されてクライアントからの利用が可能な状態になりました。

クライアントの起動は、TrialRMI Client アプリケーションをダブルクリックするだけです。こちらは GUI を持っており、起動すると Frame1 で定義したウインドウが表示されるはずです。まず、Server には、RMI サーバーを指定します。おそらく最初のテストは、サーバーとクライアントを同じコンピュータで行うと思うので、その場合は、rmi://127.0.0.1/TESTSERVER と指定されているのを確認します。別のコンピュー

タを RMI サーバーにしているのなら、そのサーバーのドメイン名やあるいは IP アドレスを指定します。そして「引数」に適当な文字を入力して、Do RMI ボタンをクリックします。1 回目の実行は少し時間がかかりますが、数秒程度待つと、返り値のテキストフィールドに、RMI サーバーの getInfo メソッドよって返された文字列が表示されるはずです。引数の文字数をレポートするはずです。

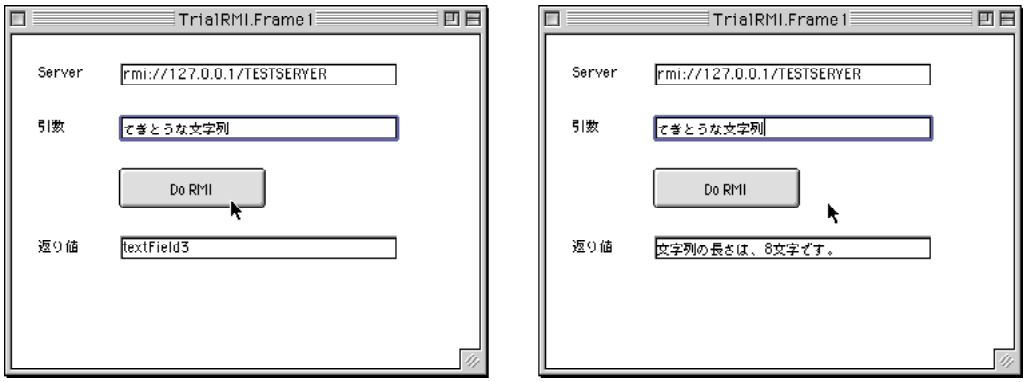

図 13-23　クライアントは引数を指定してボタンをクリックするだけ

こうしてクライアントでの実行をした後、TestServer の Java コンソールを見ると、getInfo メソッドを呼び出した履歴が見えているはずです。呼び出した時刻とそのときの引数が 1 行ずつで表示されています。

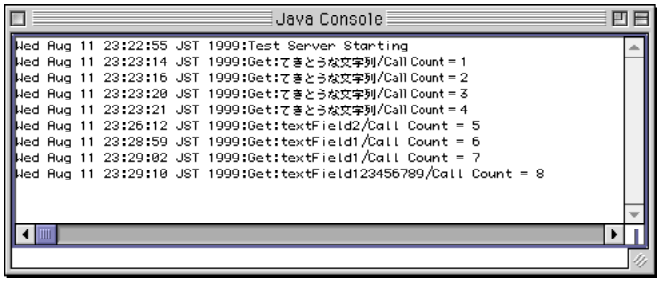

図 13-24　TestServer では呼び出し履歴を参照できる

クライアントのアプリケーションを別のコンピュータにコピーするなどして異なるマシンからクライアントを動かしてみて、TestServer のコンソールを見てみましょう。getInfo の呼び出し回数をカウントするようになっていますが、その値が順番に 1 ずつアップしており、この方法ではサーバーにある 1 つのインスタンスに対して、複数のクライアントからのメソッド呼び出しに対応していることが分かります。何も指定をしなくても、マルチユーザーに対応するという状況になっています。実際には接続し

てきたクライアントごとにスレッドを作って管理するようになっているようですが、インスタンスは1つのものを共通に利用するということになります。

もし、クライアントごとに異なるインスタンスが欲しいような場合は、2 段階になりますが、サーバー上でのインスタンスを生成する別のクラスなどを定義して、まず最初にインスタンスを生成させ、その生成したインスタンスを返すというようなメソッドを作ります。これは、ファクトリーと呼ばれる手法を用いた方法ですが、場合によってはサーバー側をこうした作り方にする必要が出てくるでしょう。

Frame1.java

クライアントのアプリケーションのウインドウ表示を行う Frame1 のソースは、ほとんどが RAD ツールによって生成されたものです。以下、ソースを掲載しておきますが、追加のポイントは前の節で説明してあるので、それを御覧ください。

```
/*
    Frame1.java

    Title:          Sample for RMI
    Author:             msyk
    Description:    Macintosh Java Report, Sample Sources
*/

package TrialRMI;

import java.awt.*;
import java.awt.event.*;
import java.rmi.*;                          //******この行を追加
import TestServer;                          //******この行を追加

public class Frame1 extends Frame
{

// IMPORTANT: Source code between BEGIN/END comment pair will be regenerated
// every time the form is saved. All manual changes will be overwritten.
// BEGIN GENERATED CODE
    // member declarations
    java.awt.Label label1 = new java.awt.Label();
    java.awt.Label label2 = new java.awt.Label();
    java.awt.Label label3 = new java.awt.Label();
    java.awt.TextField textField1 = new java.awt.TextField();
    java.awt.TextField textField2 = new java.awt.TextField();
    java.awt.TextField textField3 = new java.awt.TextField();
    java.awt.Button button1 = new java.awt.Button();
// END GENERATED CODE

    public Frame1()
```

```
    {
        System.setSecurityManager(new RMISecurityManager());
            //RMI 呼び出しができるようにセキュリティの管理を行う
    }

    public void initComponents() throws Exception
    {
// IMPORTANT: Source code between BEGIN/END comment pair will be regenerated
// every time the form is saved. All manual changes will be overwritten.
// BEGIN GENERATED CODE
        // the following code sets the frame's initial state
        label1.setText("Server");
        label1.setLocation(new java.awt.Point(20, 20));
        label1.setVisible(true);
        label1.setSize(new java.awt.Dimension(50, 20));

        label2.setText("引数");
        label2.setLocation(new java.awt.Point(20, 60));
        label2.setVisible(true);
        label2.setSize(new java.awt.Dimension(50, 20));

        label3.setText("返り値");
        label3.setLocation(new java.awt.Point(20, 150));
        label3.setVisible(true);
        label3.setSize(new java.awt.Dimension(50, 20));

        textField1.setText("rmi://127.0.0.1/TESTSERVER");
        textField1.setLocation(new java.awt.Point(80, 20));
        textField1.setVisible(true);
        textField1.setSize(new java.awt.Dimension(210, 20));

        textField2.setText("textField2");
        textField2.setLocation(new java.awt.Point(80, 60));
        textField2.setVisible(true);
        textField2.setSize(new java.awt.Dimension(210, 20));

        textField3.setText("textField3");
        textField3.setLocation(new java.awt.Point(80, 150));
        textField3.setVisible(true);
        textField3.setSize(new java.awt.Dimension(210, 20));

        button1.setLocation(new java.awt.Point(80, 100));
        button1.setLabel("Do RMI");
        button1.setVisible(true);
        button1.setSize(new java.awt.Dimension(110, 30));

        setLocation(new java.awt.Point(0, 0));
        setTitle("TrialRMI.Frame1");
        setLayout(null);
        setSize(new java.awt.Dimension(350, 235));
        add(label1);
        add(label2);
        add(label3);
        add(textField1);
```

```
            add(textField2);
            add(textField3);
            add(button1);


            button1.addActionListener(new java.awt.event.ActionListener() {
                public void actionPerformed(java.awt.event.ActionEvent e) {
                    button1ActionPerformed(e);
                }
            });

            addWindowListener(new java.awt.event.WindowAdapter() {
                public void windowClosing(java.awt.event.WindowEvent e) {
                    thisWindowClosing(e);
                }
            });
    // END GENERATED CODE
        }

        private boolean mShown = false;

        public void addNotify()
        {
            super.addNotify();

            if (mShown)
                return;

            // move components to account for insets
            Insets insets = getInsets();
            Component[] components = getComponents();
            for (int i = 0; i < components.length; i++) {
                Point location = components[i].getLocation();
                location.move(location.x, location.y + insets.top);
                components[i].setLocation(location);
            }

            mShown = true;
        }

        // Close the window when the close box is clicked
        void thisWindowClosing(java.awt.event.WindowEvent e)
        {
            setVisible(false);
            dispose();
            System.exit(0);
        }

        //ボタンをクリックした時のメソッド
        public void button1ActionPerformed(java.awt.event.ActionEvent e)
        {
            try    {
                TestServer rs = (TestServer)Naming.lookup(textField1.getText());
                    //textField3 の内容から、指定したクラスを探し、
```

```
                //      そのクラスのインスタンスへの参照を得る
            String r = rs.getInfo(textField2.getText());
                //サーバーにあるオブジェクトの getInfo メソッドを呼び出した。
                //      引数は textField1 の値
            textField3.setText(r);
                //返り値を textField2 に設定した
        }
        catch(Exception ex)    {
            System.out.println(ex.getMessage());
        }
    }

}
```

Application1.java

クライアントアプリケーションを、アプリケーションとして働かせる Application1 クラスは、Java アプリケーションウィザードによって生成されたそのままを利用しました。念のため、ソースを掲載しておきます。

```
/*
    Application1.java

    Title:          Sample for RMI
    Author:             msyk
    Description:    Macintosh Java Report, Sample Sources
*/

package TrialRMI;

public class Application1
{
    public Application1()
    {
        try {
            Frame1 frame = new Frame1();
            frame.initComponents();
            frame.setVisible(true);
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }

    // Main entry point
    static public void main(String[] args)
    {
        new Application1();
    }
```

```
}
```

（第 13 章以上）

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z

あ

い

う

え

お

か

き

く

け

こ

さ

し

す

な

に

ね

の

は

ひ

ふ

へ

ほ

ま

む

め

も

ゆ

よ

ら

り

る

れ

ろ

わ