Net Express

オブジェクト指向プログラミング

Micro Focus NetExpress™

オブジェクト指向プログラミング

Micro Focus®

第 4 版
1998 年 10 月

序文

本書、*オブジェクト指向プログラミング* は、Micro Focus NetExpress 製品に 添付のマニュアルのうちの一部です。*オブジェクト指向プログラミング* は、HTML形式で提供され、 Object COBOLによるプログラミングをできるだけ早く始められるようにすることを目的としています。

目次

第1章　オブジェクト指向プログラミングを始める前に

本書は、Object COBOL のオブジェクト指向 (OO) 機能を使ってすぐにプログラミングを開始できるように設計されています。 OO 機能を使う基本的な方法は 2 通りあります。

- 内部手続き型 COBOL コードから、他のオブジェクトを使ってオブジェクトを作成することができます。OLE オートメーション サポートを使って、Windows システム上の非 COBOL オブジェクトを利用することも含みます。
- ユーザ独自のクラスおよびオブジェクトを開発して、新規の OO アプリケーションを作成することができます。OO 分析や設計技術を学習したい方のために、本書では関連する詳細な情報を検索できます。

多くの場合、初めはオブジェクトを使い、必要に応じて独自に開発することになります。

Object COBOL の OO 機能を初めてご使用になる場合は、次の手順に従って本書をお読みになることをお勧めします。

1. 新しい概念に慣れる
    - 第 1 章 オブジェクト指向プログラミングを始める前に
    - 第 2 章 Object COBOLの概念
2. チュートリアルを使って作業する
    - 第 3 章 チュートリアルの概要
    - 第 4 章 オブジェクトとメッセージのチュートリアル
    - 第 5 章 クラス プログラムを作るチュートリアル
    - 第 6 章 継承のチュートリアル
    - 第 7 章 コレクション、固有値、ディクショナリのチュートリアル
    - 第 8 章 例外取り扱いのチュートリアル
    - 第 9 章　GUI プログラミングのチュートリアル
    - 第 10 章 要件ベースのボキャブラリのチュートリアル
3. クラス ライブラリのフレーム ワークおよび提供されている機能について詳細を知る

- 第 13 章 クラス ライブラリの紹介
- 第 14 章 コレクションのフレームワーク
- 第 15 章 固有データ
- 第 16 章 CallBack のフレームワーク
- 第 17 章 例外取り扱いのフレームワーク
- 第 18 章 GUI クラスのフレームワーク
- 第 19 章 コンポーネントのフレームワーク

4. プログラミングを開始する
    - 第 11 章 プログラムでのオブジェクトの使い方
    - 第 12 章 クラス プログラム

1.1 Object COBOL の OO 機能

Object COBOL を使うと、COBOL のオブジェクト指向プログラミングが可能になります。一方、COBOL の元の構文は保持されます。OO を使わない場合は、Micro Focus COBOL を使ったときと同じ方法で Micro Focus Object COBOL を使うことができます。

OO プログラムを COBOLで書けるようにするため、Micro Focus は次のコンポーネントおよび機能を COBOL システムに追加しています。

- OO の新規 COBOL 構文のコンパイラ サポート

  Object COBOL には新規構文があります。新規の文（INVOKE）が 1 つ、新規のデータ 型（OBJECT REFERENCE）が 1 つ、および見出し段落の新しい型が 3 つです。

- Object COBOL ランタイム システム

  オブジェクト、動的メッセージ バインド、および継承を COBOL ランタイム システムに作成するようサポートします。

- Object COBOL クラス ライブラリ

  独自のアプリケーションをビルドするための基礎として使う、あらかじめ定義されたオブジェクトのライブラリです。

- 統合開発環境の一部としてのブラウズ ウィンドウ

アプリケーション内でクラスとメソッドの間を簡単に移動できます。

1.2 オブジェクト指向について学ぶには

Object COBOL に添付のマニュアルでは、Object COBOL でプログラミングを開始する方法を説明します。これによって、他の人が書いたオブジェクトを使用したり、OLE オートメーションを使用する方法がわかります。しかし、OO が初めての方が OO システムを設計したい場合は、オブジェクト指向デザイン (OOD) およびオブジェクト指向分析 (OOA) の原理について学習すると、Object COBOL を最大限に利用できるようにします。

言語に依存しない OOD および OOA を行うトレーニングもあります。本書では、以下について説明します。

- 参考資料
- コンピュータベースのトレーニング

1.2.1 参考資料

オブジェクト指向の方法や技術に関する文献としては、次のようなものがあります。

Grady Booch 著
『*Object-Oriented Design*』
Benjamin/Cummings, 1994 ISBN: 0-8053-0091-0

Ivor Jacobson 著
『*Object-Oriented Software Engineering*』
Addison-Wesley, 1992 ISBN: 0-201-54435-0

James Rumbaugh 著
『*Object-Oriented Modeling and Design*』
Prentice Hall, 1991 ISBN: 0-13-629841-9

Raymond Obin 著
『*Object Orientation ・An Introduction for COBOL Programmers*』
Micro Focus Publishing, 1993 ISBN 1-56928-005-3
オブジェクト指向について学習したい COBOL プログラマを対象としています。今後の COBOL 言語への ANSI 拡張の概要についても説明しています。

Sally Shlaer and Steve Mellor 共著
『*Object-Oriented Systems Analysis: Modeling the World in Data* 』
Prentice Hall, 1988 ISBN: 0-13-629023-1

『*Object Lifecycles: Modeling the World in States* 』

Prentice Hall, 1992 ISBN: 0-13-629940-7

David Taylor 著 『*Object Oriented Information Systems: Planning and Implementation*』
John Wiley, ISBN: 0-471-54364-0
システム インストールおよび開発を成功させるために決定を下す必要のある管理者を対象としています。

Rebecca Wirfs-Brock and B. Wilkerson 共著
『*Designing Object-oriented Software*』

1.2.2 コンピュータ ベースのトレーニング

次のコースは、オブジェクト管理研究所で開発されました。Micro Focus のマニュアルから利用できます。

- 『Object Technology for Managers』
  オブジェクト指向の分析・設計の概要を知る必要のある管理者および役員を対象としています。

- 『The OOA/OOD Series』
  オブジェクト指向のプロジェクトに関係するプログラマとテクニカル マネージャを対象としています。3つのコースがあります。

- 『Object COBOL』
  COBOL を使って OO プログラミングを学習したいプログラマを対象としています。

第2章　Object COBOLの概念

オブジェクト指向プログラミングでは、次のような新しい考え方や概念を取り入れています。この章では、これらの概念が Object COBOL でどのように実現されているかについて説明します。Object COBOLを OO 言語として効果的に使用するには、新しい考え方や概念がどのように実現されているかを理解する必要があります。

- オブジェクト
  オブジェクトについては、2.1 オブジェクトを参照してください。
- クラス
  クラスについては、2.2 クラスを参照してください。
- メソッド
  メソッドについては、2.3 メソッドを参照してください。
- 継承
  継承については、2.4 継承を参照してください。
- 多相性
  多相性については、2.5 多相性を参照してください。
- メッセージと結合
  メッセージと結合については、2.6 メッセージと結合を参照してください。

2.1 オブジェクト

オブジェクトは、データと、データに実行される手続きとをカプセル化したものです。データはオブジェクトの属性、手続きはオブジェクトのメソッドと呼びます。オブジェクト指向アプリケーションでは、すべてのオブジェクトが固有のオブジェクト識別子を持っています。オブジェクト識別子はオブジェクトの作成時に割り当てられます。これをインスタンス化といいます。

オブジェクト指向アプリケーションにおける大部分のオブジェクトは、実世界のオブジェクトを表しています。例えば、金融システムの世界では、顧客、口座、元帳などを表すオブジェクトが考えられます。口座の属性には残高、メソッドには Debit（借方）、Credit（貸方）、GetBalance（残高獲得）などのメソッドが考えられます。図 2-1 は、このオブジェクトを 2 つの方法で示しています。

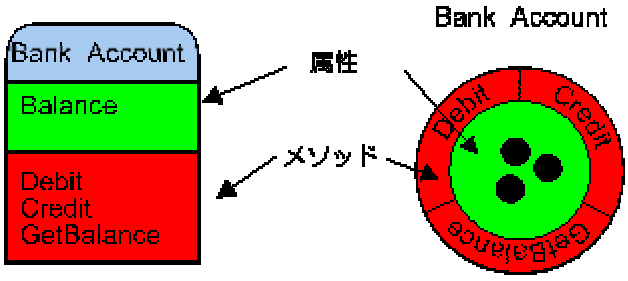


図 2-1 オブジェクト

オブジェクトのユーザが、オブジェクトの属性を調べたり属性を変更するには、そのことをオブジェクトに要求しなければなりません。このような要求をメッセージと呼びます。各メッセージによって、オブジェクトがサポートするメソッドが呼び出されます。オブジェクト インタフェースは、オブジェクトが応答するすべてのメッセージを記述したものです。

口座の残高を調べるには、Account オブジェクトにメッセージ GetBalance を送ります。実際のデータ表現は、Account オブジェクトしか知りません。オブジェクトインタフェースが同じなら、システムの他の部分に影響を与えずにオブジェクトの表現方法やデータの操作方法の内部記述を変えることができます。データを、常に期待されるフォーマットで提供するのは GetBalance の役割です。

関連項目

- 2.1 オブジェクト
- 第 11 章: プログラムでのオブジェクトの使い方

2.2 クラス

クラスはオブジェクトを作成するためのテンプレートです。クラスには、特定の型のオブジェクトを作成するのに必要なすべての情報が入っています。Account クラスは Account オブジェクト、Ledger（元帳）クラスは Ledger オブジェクトを作成するのに使用します。Account オブジェクトは Account クラスのインスタンスであるといいます。

Object COBOL におけるクラスは Object COBOL プログラムで、入れ子のプログラムのセットからできています。実行時にクラスがロードされると、OO RTS はクラス オブジェクトを作成し、それがクラスを表します。クラス オブジェクトによって、クラスのインスタンスを作成することができます。

Object COBOL のクラスでは、外側のプログラムにクラス オブジェクトに固有なすべての属性とメソッドが入っています。オブジェクト プログラムには、各インスタンスオブジェクトに固有な属性とメソッドが入っています。クラスの構造を図 2-2 に示します。

図 2-2 クラス プログラム

Shared data & definitions (共有データと定義)	共有データと定義は、クラスプログラムのすべての部分にアクセスすることができます。データはクラスプログラム データ部で宣言されます。定義には CLASS-ID 見出しのデータ継承のための設定、環境部の設定が含まれ、CLASS-CONTROL段落で登録されたすべてのクラスが入れられます。
Class initialization code (クラス初期化コード)	クラス オブジェクトの中に任意に「initializeClass」メソッドを含むことができます。これはクラスが初めて読み込まれたときに実行され（アプリケーションが実行されるたびに一度だけ）、任意の初期設定コードのために使用することができます。
Class Object (クラス オブジェクト)	クラス オブジェクトは CLASS-OBJECT 見出しで始まり、END CLASS-OBJECT 見出しで終わる入れ子のプログラムです。クラス データを宣言するための Object-storage 節と 1 つまたは複数のクラス メソッドを任意に含むことができます。クラス プログラム内にクラス オブジェクトを含む必要はありません。一部のクラスはインスタンス動作のみを定義します。
Object (オブジェクト)	インスタンス オブジェクトは OBJECT 見出しで始まり、END OBJECT 見出しで終わる入れ子のプログラムです。インスタンス データを宣言するための Object-storage 節と 1 つまたは複数のインスタンス メソッドを任意に含むことができます。クラス プログラム内にインスタンス オブジェクトを含む必要はありません。一部のクラスはクラス動作のみを定義します。

クラス オブジェクトはオブジェクト自身で、サービスのセットを提供し、属性を保持することができます。クラスはオブジェクトですが、それによって作成されるインスタンスと同じ動作をするわけではありません。

クラスは印刷に使う原版に似ています。1 つの原版からは同じフォームが何枚でも印刷できます。このように原版からはフォームが印刷できますが、原版はフォーム自身ではありません。Object COBOL のクラスプログラムには、クラスとオブジェクトの両方のコードとデータが入っています。Object COBOL のランタイム コードでは、クラスのインスタンスをたくさん作ることができます（図 2-3 を参照）。

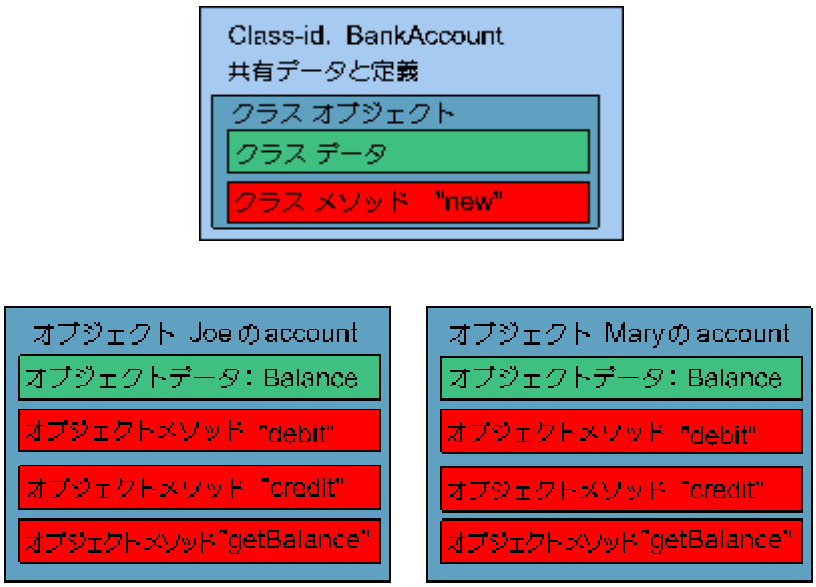


図 2-3 クラスのインスタンスの作成

各インスタンスは、実行単位の中の独立したプログラムのようなものです。しかし、各オブジェクトごとにメモリ内に別々に存在するのはオブジェクト データだけで、コードはインスタンス間で共有されます（図 2-4 を参照）。

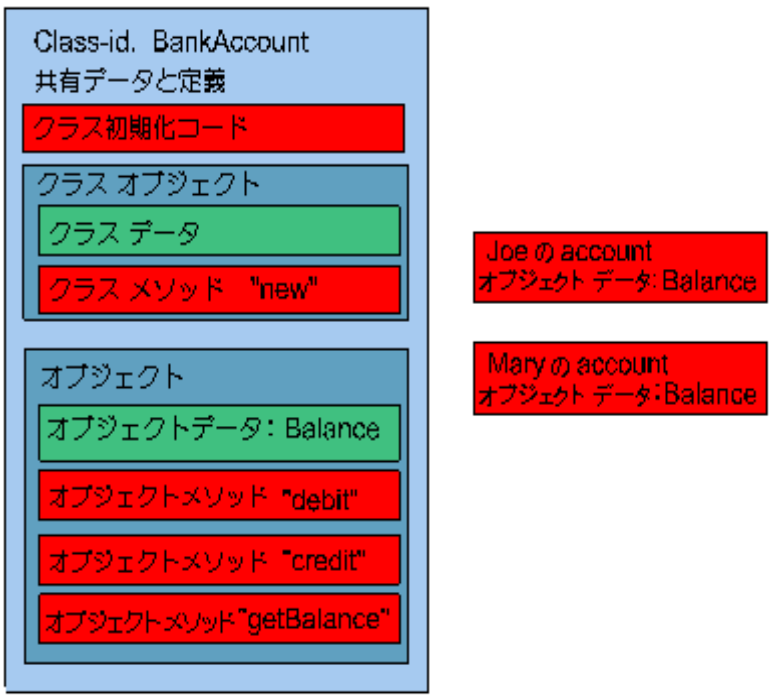


図 2-4 コードはすべてのインスタンスによって共有される

関連項目

- 2.2 クラス
- 第 12 章: クラス プログラム
- 12.2.1 クラス プログラムの構造

2.3 メソッド

メソッドは、オブジェクトの動作を実現するためのコードです。Object COBOL の各メソッドは、クラスまたはオブジェクト プログラム内で入れ子にされた個別のプログラムです。インスタンス メソッドは、自身のメソッドデータ、インスタンス データ、クラス プログラムのデータ部で宣言された共有データにアクセスできます。クラス メソッドは、自身のデータ、クラスデータと共有データにアクセスできます。

クラス データとインスタンス データの実現は、Micro Focus Object COBOL と OO COBOL に関する ANSI 標準規格とでは異なっています。ANSI ではオブジェクトデータを意味するために working-storage 節を再定義しています。また、クラスと全インスタンス間におけるデータ共有という概念はありません。Micro Focus Object COBOL は、新しい節 Object-Storage を使用して、クラスとインスタンス データを宣言します。

Micro Focus コンパイラでは、object-storage 節の代わりに working-storage 節内でクラスとインスタンス データを宣言し、ANSI 標準と同様に動作させることができます。この場合は、コンパイラ指令 OOCTRL(+W) をセットしてください。

2.4 継承

クラス間に共通の特性がある場合、継承によってコードを再利用できます。前に示した金融の例では、普通預金、高金利預金、当座預金などの口座があります。そして、これらすべての口座に共通する特徴があります。すなわち、残高属性、メソッド Credit と GetBalance です。一方違いもあります。普通預金では金利が支払われますが、当座預金では借越しが許されます。

Account クラスは、そのすべてのサブクラスに共通のメソッドと属性を実現します。サブクラスは、それぞれに固有の属性とメソッドを実現します。この例では、Account クラスは抽象クラスで、サブクラスの動作を提供するだけです。Account クラスのインスタンスを実際に作成することはありません。

Deposit Account は、貸越しを許さない独自バージョンの Debit を実現します。銀行口座に関する継承階層の例を図 2-5 に示します。

継承を使うと、後でシステムを更新したい場合も新しいサブクラスを追加するだけですみます。例えば、2 種類の新しい普通預金を導入するとします。1 つはすぐに引き出せるものと、もう 1 つは引き出しを通知しなければならない代わりに利子が高いものです。これらは、新しい動作を追加することによって SavingsAccount クラスからサブクラス化できます。これらのサブクラスも、すべての口座オブジェクトが使用するメッセージインタフェースに対

して応答するため、変更によってシステムの残りの部分が影響を受けることはありません。

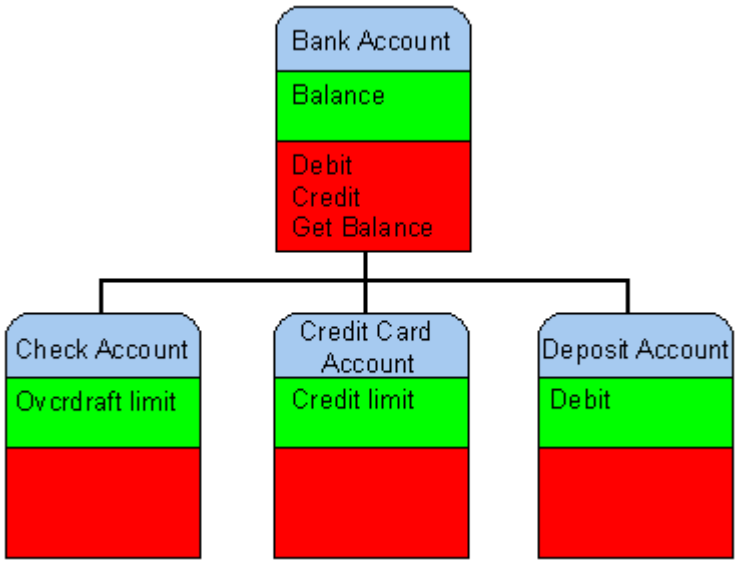


図 2-5 継承

Object COBOL にはクラス ライブラリがあります。クラス ライブラリの中では、すべてのクラスがクラス Base を祖先としています。Base は、すべてのクラスに必要なメソッドを提供します。これには、オブジェクトの作成や廃棄のためのメソッドなどがあります。独自に作成するクラスは Base のサブクラスであるのが普通です。

Object COBOL のサブクラスは、その親クラス（スーパークラス）のすべてのクラス メソッドにアクセスできます。サブクラスのインスタンスは、スーパークラスのすべてのオブジェクト メソッドにアクセスできます。CLASS-ID 見出し内の句によってデータも継承できます。

関連項目

- 12.3.1 メソッドの継承方法
- 12.3.2 データの継承方法
- 2.4 継承

2.5 多相性

多相性は、すべてのオブジェクト指向プログラミング システムにとって重要な部分です。多相性とは、複数のオブジェクトに同じメッセージを送ることによって、それぞれ異なるメソッドを呼び出せることを意味します。

例えば、四角形を表すオブジェクトと円を表すオブジェクトからなるグラフィック描画システムについて考えてみます。四角形と円を描くためのメソッドは異なりますが、ディスプレイ コントローラは、任意のグラフィカル オブジェクトに対して、その種類に関係なくメッセージ Draw を送ることができます。メッセージの受け手は、それぞれ自分の Draw メソッドを実行して正しい結果を生成します（図 2-6 を参照）。

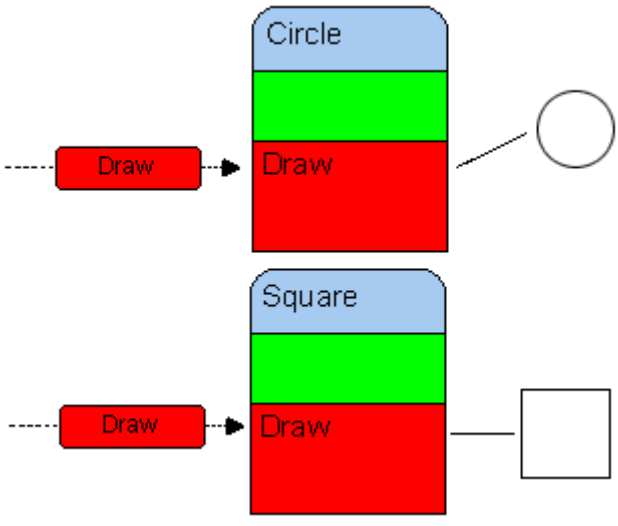


図 2-6 多相性

2.6 メッセージと結合

前に説明したように、メッセージはオブジェクトに対してサービスの実行を要求するための手段です。メッセージは、必ず次のものから構成されます。

- オブジェクト参照
- メソッド セレクタ

メッセージには、任意で入力パラメータや出力パラメータを指定することができます。出力パラメータがある場合、送り手はメッセージに対する応答を期待します。

ランタイム システムは、オブジェクト参照によってメッセージが意図するオブジェクトを見つけることができます。メッセージのターゲットは受け手と呼ばれます。Object COBOL のランタイム システムは、動的結合を使ってメッセージの受け手がどれかを判断します。つまり、メッセージの受け手はコンパイル時ではなく実行時に判断されます。

メソッド セレクタはメッセージのテキストです。これは、受け手に対してどのメソッドを呼び出すかを指示します。Object COBOL では、INVOKE という新しい文を使ってメッセージを送ります。図 2-7 はメッセージのコンポーネントを図と Object COBOL の構文で示しています。

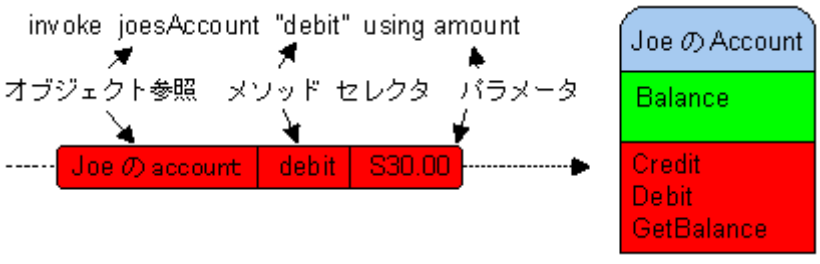


図 2-7 メッセージ

関連項目

- 2.1 オブジェクト
- 2.6 メッセージと結合
- 11.3 メッセージの送信
- 12.6.3 自分宛てのメッセージ送信

第3章　チュートリアルの概要

OO チュートリアルは、Object COBOL での OO プログラミングを学習するために役立ちます。これらのチュートリアルでは、Object COBOL の Animator を使用して、作業コードの例を説明し、Object COBOL と Object COBOL クラス ライブラリでのさまざまなプログラミングを示しています。

本書には次のようなチュートリアルがあります。

- 第 4 章 オブジェクトとメッセージのチュートリアル
- 第 5 章 クラス プログラムを作るチュートリアル
- 第 6 章 継承のチュートリアル
- 第 7 章 コレクション、固有値、ディクショナリのチュートリアル
- 第 8 章 例外取り扱いのチュートリアル
- 第 9 章 GUI プログラミングのチュートリアル (Phonebook)
- 第 10 章 要件ベースのボキャブラリのチュートリアル

第4章　オブジェクトとメッセージのチュートリアル

チュートリアルは、クラスといくつかの簡単な動作を使った NetExpress プログラミングの入門です。このクラスを Parrot（おうむ）クラスと呼ぶことにします。チュートリアルでは、Parrot クラスのインスタンスを作成します。Parrot クラスのインスタンスは Parrot オブジェクトで、作成した Parrot にはメッセージを送ることができます。 チュートリアルは次のセッションからできています。

1. Parrot クラスのインタフェース
2. メッセージの送信
3. まとめ

学習時間：20分

4.1 Parrot クラスのインタフェース

この節では、Parrot クラスを紹介し、その動作とパブリックインタフェースについて説明します。 Parrot のためのインタフェースは次の 2 つに分けられます。

- Parrot クラス インタフェース（クラス メソッド）
- Parrot インスタンス インタフェース（インスタンス メソッド）

Parrot クラスのインスタンスはデスクトップに表示されるウィンドウです。Parrot ウィンドウには順番に番号が付いています。各番号はタイトル バーに表示されます。デスクトップから Parrot ウィンドウのサイズを変更したり閉じたりすることはできません。Parrot ウィンドウは、それを作成したアプリケーションによってだけ制御されます。Parrot インスタンスは、理解できないメッセージをすべてトラップし、エコー バックします。オブジェクトが理解できないメッセージを受け取った場合、デフォルト時の動作ではエラー メッセージを表示し、アプリケーションを終了します。

Parrot クラスのパブリック オブジェクト インタフェースを次に示します。

クラスメソッド	説明
"new" returning anObject	Parrot クラスのインスタンスを anObject 内の参照と共に返します。
"howMany" returning aNumber	これまでに作成された parrot インスタンスの数を返します。

インスタンス メソッド	説明
"sayHello" using aName	Parrot ウィンドウの Response フィールドに "**Hello aName**" を表示します。aName 内の文字列の最後はスペースでなければなりません。
"whatNumber" returning aNumber	この parrot インスタンスの数を返します（すべての parrot は作られるたびに順番に番号が付けられます）。

これらのメソッドに加え、Parrot クラスはそのスーパークラス（Window）によって実現されるすべてのメソッドを継承します。スーパークラスは、このクラスが基本動作を継承している親です。Parrot クラスはWindowのすべての クラス メソッド を継承し、そのインスタンス は Window のすべてのインスタンス メソッドを継承します。つまり Parrot は、自身のメッセージとともに Window のパブリック インタフェースで与えられるすべてのメッセージに対して応答します。 Window クラスは NetExpress のクラス ライブラリに含まれています。

次のセッションに進んでください。次は、NetExpress デバッガを使ったメッセージの送信について説明します。

4.2 メッセージの送信

この節では Parrot クラスのインスタンスを作成し、それにメッセージを送ります。送信には、あらかじめ用意されたプログラム hello.cbl を使用します。 プログラム hello.cbl は Object COBOL クラスではなく、Object COBOL のオブジェクトを使用する手続きコードです。hello.cbl は次の Object COBOL 構文を使ってオブジェクトと通信します。

- class-control 段落
  プログラムが使用するクラスを宣言し、ファイル名とリンクします。
- object reference データ型
  オブジェクト参照を入れるための変数を宣言します。
- invoke 文
  オブジェクトにメッセージを送ります。

オブジェクトの簡単な動作をいくつか見るために hello をアニメートします。コードは次のようなデモンストレーションを行います。

- クラス オブジェクトとそれによって作成されたインスタンス オブジェクトが、各種のメッセージに応答する
- 作成された各インスタンスが、自身の固有なインスタンス データを持っている
- クラス オブジェクトはクラス データを持っており、それは固有で、そのインスタンスからは直接使用で

きない

- オブジェクトのデータに対するアクセスは、パブリック インタフェースを通じて（メッセージを送ることによって）行う。オブジェクトがメソッドを通じて明示的にアクセスを与えていないものに対しては、データの読み込みや変更はできない

4.3 hello のアニメート

1. NetExpress 統合開発環境を開始して、Parrot チュートリアルプロジェクト (parrot.app) をロードします。

   このプロジェクトには、実行する予定のプログラム hello.cbl が含まれます。

2. [プロジェクト] メニューの [リビルド] をクリックして、デバッグ用に hello.cbl をコンパイルします。

3. [アニメート] メニューの [ステップ実行] をクリックして、hello.cbl のアニメートを起動します。

   最初の文 **call "apigui"** は、実行に備えて強調表示されています。

4. 最初の文をステップ実行します。

   このアプリケーションによって使用される GUI クラス ライブラリがロードされます。

5. 次の行のコード (**invoke Parrot "new" ...**) をステップ実行します。

   Parrot クラスにメッセージ "new" が送られます。Parrot クラスは自身のインスタンスを作成し、それに対するオブジェクトハンドルを aParrot によって返します。 新しいインスタンスは、デスクトップ上に Parrot 01 ウィンドウを開きます。

6. 「検査」機能を使って、データ項目 aParrot の内容を検討します。

   このデータ項目は、object reference を使って宣言します。これは、Object COBOL プログラミングのために取り入れられた新しい COBOL データ型です。これを検討すると、作成した新しいオブジェクトに割り当てられた 4 バイトのオブジェクト ハンドルを確認できます。

   Object COBOL のプログラミングでは、オブジェクトにメッセージを送るためのアドレスとして、オブジェクト ハンドルを使用します。 オブジェクト ハンドルは、異なる複数のプログラムまたはオブジェクトの間でパラメータとして受け渡しできます。このため、アプリケーションの各部の間でオブジェクトの受け渡し渡しができるます。

7. APARROT モニタウィンドウを閉じます。

8. タグ H003 の下の行をステップ実行します。(**invoke aParrot "sayHello"**).

   Parrot 01 は、応答フィールドに "Hello ポリー" を、メッセージ フィールドに "sayHello" を表示します。

Parrots は、必ず送信されたメッセージを繰り返します。

9. タグ H004 の下の行をステップ実行します (**invoke aParrot "saySomethingElse"**)。

   Parrots は、このメッセージを理解できません。"**saySomethingElse**" は、Parrot クラス、またはそのスーパークラスによって実現されるメソッドではありません。 parrot はこのエラーをトラップし、理解できなかったメッセージを再表示します。オブジェクトに理解できないメッセージを送ったときのデフォルト動作では、エラーメッセージが表示されてアプリケーションが終了します。 Parrot クラスプログラムは、オブジェクトが他の場合に Base (クラス ライブラリの一部) から継承する "**doesNotUnderstand**" メソッドを再実現することによって、この動作を無視します。

10. タグ H005 の下の行をステップ実行します (**invoke aParrot "whatNumber"**)。

    parrot の ID が parrotNumber に返されます。このデータ項目を調べると、1 の値を持つことがわかります。

11. タグ H006 の下の行をステップ実行します。

    Parrot クラスの新しいインスタンスが作成されて、オブジェクト ハンドルが aParrot に入れられます。最初に作成した parrot である Parrot 01 は、まだ存在しています。しかし、オブジェクト ハンドルを与えた唯一のデータ項目をここで上書きしたため、この項目に対してメッセージを送ることはできなくなります。

    これは、Object COBOL でのプログラミングで特に注意しなければならない点です。オブジェクトに対するオブジェクト ハンドルを失うと、そのオブジェクトにメッセージを送ったり、廃棄したりするための簡単な方法がなくなります。 クラス オブジェクトにメッセージ "**allInstances**" を送ることによって、特定クラスのインスタンスに関するすべてのハンドルのコレクションを取得できます。 しかし、実際のほとんどのアプリケーションでは、個々の各インスタンスを誰が作成したかはわからず、どれを安全に廃棄できるかを知る方法はありません。

12. タグ H007 の下の行をステップ実行します。

    Parrot の新しいインスタンスにメッセージが送信されます。

13. タグ H008 の下の行をステップ実行します。

    Parrot のクラス データが照会されます。

    parrotNumber を検討すると、値 2 が含まれることを確認できます。これは、ここまでに作成された parrot インスタンスの数です。

14. タグ H009 の下の行をステップ実行します。

    "**finalize**" メッセージをオブジェクトに送信すると、その記憶域が割り当て解除されます。これを「最終

化する」といいます。

アプリケーションで終了したオブジェクトは、最終化する必要があります。オブジェクトを最終化したら、そのオブジェクト ハンドルにメッセージを送信してはなりません。結果が予測不能になります。オブジェクト指向 COBOL に提案された ANSI 標準には、自動ガーベージ コレクション (automatic garbage-collection) が含まれます。 ガーベージ コレクションでは、使用されなくなったオブジェクトをランタイム システムが廃棄します。ANSI 標準が固定されて、Micro Focus がこの機能を実現した時には、オブジェクトを最終化するためのコードを入れる必要はなくなります。

これで、hello.cbl の実行は終わりです。

4.4 まとめ

これでオブジェクトの作成とメッセージの送信に関するチュートリアルが終了しました。ここでは、次のことを学習してきました。

- オブジェクトを作成する方法
- メッセージを送る方法
- オブジェクトを仕上げる方法

第5章　クラス プログラムを作るチュートリアル

オブジェクトとメッセージのチュートリアルでは、クラスのインスタンスを作成する方法とメッセージを送る方法について学びました。この章では、オブジェクトのためのクラスを独自に作る方法について学びます。

ここでは BeepWindow というクラスを作成します。これは Window のサブクラスです。したがって、BeepWindow はクラス ライブラリ Window クラスにある、すべての基本動作を継承します。

BeepWindow のインスタンスは、1 つのプッシュボタンを持ったウィンドウです。ボタンを押すと警告音が鳴ります。

最初に、クラス プログラムがどのように組み立てられているかを見ます。これには、新しい予約語のある、ANSI 85 COBOL に類似した構文を使います。それに続く各節では、BeepWindow クラスを順番に組み立てるコードをアニメートします。

このチュートリアルは、次のセッションからできています。

1. クラス プログラムの構造
2. ウィンドウの作成
3. イベントの処理
4. まとめ

学習時間：30分

5.1 クラス プログラムの構造

このチュートリアルでは、最初にクラス プログラムの全体構造を見ます。これには、NetExpress の Browse 機能を使って BeepWindow の構造を調べます。クラスプログラムは複数の入れ子のプログラムが集まってできています。プログラムの入れ子は、ANSI 85 標準の COBOL で取り入れられた概念です。

クラス構造について、次の順に説明します。

- クラスの識別
- プログラムによって使用されるクラスの識別
- 共有データの宣言
- クラス オブジェクト
- クラス オブジェクト データの宣言

- クラスメソッドの作成
- オブジェクトプログラム
- インスタンスメソッド

処理を進める前に、BeepWindow クラスを NetExpress 統合開発環境にロードしてください。

BeepWindow のロード方法

1. NetExpress 統合開発環境を開始して、Objects プロジェクト (beepwin.app) をロードします。

   このプロジェクトには、プログラム beepwin.cbl と関連のトリガ プログラム、beeptrig.cbl が含まれます。ここでは、これらのプログラムを学習して実行します。

2. [プロジェクト] メニューの [リビルド] をクリックして、デバッグ用にプロジェクト ファイルをコンパイルします。

3. [検索] メニューの [ブラウザ] をクリックして、[ブラウザ] ウィンドウを表示します。

4. [ブラウザ] ペインの左側にある [Window] の隣のプラス記号 (+) をクリックして、参照する BeepWindow を選択してから、左側の [ブラウザ] ペインで BeepWindow を選択します。

5. [ブラウザ] ペインの一番上にあるプッシュボタンをクリックして、[ブラウザ] ウィンドウにメソッドだけを表示し、[プログラム] ボタンだけが押されている状態で示します。

5.1.1 クラスの識別

クラス プログラムは class-id 識別子で始まり、end class 句で終わります。これらによって一番外側の入れ子が囲まれます。BeepWindow クラスは次のようになります。

```
class-id. BeepWindow

          data is protected

          inherits from Window.

 ...

 end class BeepWindow.
```

inherits from 句は BeepWindow のスーパークラスである Window を識別します。data is protected 句は、BeepWindow の任意のサブクラスが BeepWindow のデータを継承できるようにします。この句を省略するか、data is private に置き換えた場合、BeepWindow のサブクラスは、継承されたデータを直接にはアクセスできません。

BeepWindow 識別コードの表示

1. BeepWindow とマークされた BeepWindow のコードを表示します。 class-id 見出しは、プログラムの開始を示すため、右側の [ブラウザ] ペインで BeepWindow をクリックすると、メソッドとして示されます。

   編集ウィンドウには、赤でアウトラインされた class-id 見出しが示されます。

2. end class 見出しは、プログラムの開始をマークしないため、[ブラウザ] ウィンドウから直接選択することができません。この見出しを表示するには、編集ウィンドウのテキストの最後に進んでください。

   end class 見出しの class-name は、class-id 見出しの class-name に一致しなければなりません。一致しないと、コンパイラによって構文エラーのフラグが付けられます。

5.1.2 プログラムによって使用されるクラスの識別

class-control段落は、プログラムによって使用されるクラスを実現する実行可能コードファイルを識別します。スーパークラス、クラス自身、クラスから呼び出されるすべてのクラスは、class-control の中で識別する必要があります。class-control 段落は次のようになっています。

```
class-control.

    Window is class "window"

    BeepWindow is class "beepwin1"

    ...

    .
```

class 句には次の 2 つの目的があります。

- 指定された各クラスについてデータ項目を作成します。このデータ項目には、実行時にクラス オブジェクトに対するオブジェクト ハンドルが入れられます。
- NetExpress のランタイムが、クラス プログラムの実行可能コードによってファイルを検索およびロードできるようにします。クラス オブジェクトが最初のメッセージを受け取る前に、必ずクラスがロードされます。これは、一般にクラス オブジェクトに最初のメッセージを送った後で、クラス オブジェクトがそれを受け取る前に起こります。

BeepWindow の class-control 段落の表示

1. [ブラウザ] ウィンドウから BeepWindow クラスを選択します。
2. 次の文が表示されるまで、編集ウィンドウを数行下にスクロールします。

```
class-control.
```

BeepWindow が使用するすべてのクラスが表示されます。BeepWindow とそのスーパークラスである Window も、表示されています。

5.1.3 共有データの宣言

クラスの一番上にあるData Division (データ部) のデータはすべて、共有データです。つまり、クラスとインスタンス オブジェクトの両方、およびすべてのクラス初期化コードに使用できます。クラス プログラムのデータ部にある、ANSI '85 COBOL プログラムで使用できるものと同じ節すべてを使用できます。

BeepWindow クラスは、この節だけを使用して、レベル 78 データ項目を宣言するコピーファイルをインクルードします。レベル-78 データ項目は、実際にはプログラム内の記憶域スペースをまったく占有しない定数であるため、各オブジェクトに記憶域オーバーヘッドを課すことなく、インスタンス データの一部として宣言されている場合があります。 ここでは、これらのデータ項目は、この節の使用法を示すためにだけ宣言されています。

注記 :

Working-Storage のこの使用法は、Micro Focus の機能であり、現在、OO COBOL に提案されている ANSI 標準の一部ではありません。

BeepWindow の共有データの表示

1. BeepWindow の唯一の共有データは、working-storage 節にあります。これは、このチュートリアルの最初に検討した class-control 段落の下にあります。 見つかるまで編集ウィンドウを下にスクロールしてください。
2. ここにあるデータは、コピーファイル p2cevent.cpy からのレベル 78 データだけです。このデータは、[表示] メニューの [表示/非表示] をクリックすることにより、IDE のコピーファイル表示/非表示機能を使用して表示、または非表示にすることができます。

5.1.4 クラス オブジェクト

クラス オブジェクト プログラムは、クラス オブジェクトのデータとメソッドを定義します。これは、クラス プログラム内に入れ子にされ、次のようになります。

```
class-object.

      object-storage section.

* class data .
```

```
 ...
* class methods
 end class-object.
```

エディタ ウィンドウを下方スクロールすると、BeepWindow のクラス オブジェクト コードを表示できます。BeepWindow は、非常に単純なクラスであり、実際には独自のクラス動作を何も定義しません。必要なものはすべて、スーパークラスである Window から継承しています。 ただし、class-object 見出しをインクルードしたため、クラス内での位置を表示できます。

5.1.5 クラス オブジェクト データの宣言

クラス オブジェクト内の Object-Storage 節は、クラス オブジェクト データを定義します。クラス オブジェクト データは、クラス メソッドからしかアクセスできません。これは、 直接アクセスのために、サブクラスによっても継承できます（これは、Class-Id 段落の内容によって決まります）。BeepWindow は、クラス オブジェクト データをまったく宣言しませんが、object-storage 節見出しがインクルードされているため、必要な場合はどこに行くかを確認できます。

5.1.6 クラスメソッドの作成

クラスメソッドは、入れ子のプログラムです。次のコードは、BeepWindow のための "new" メソッドの概要を示しています。

```
method-id. "new".
data division.
* メソッドに必要なデータ
 ...
 linkage section.
01 aBeepWindow          object reference.
01 aParent              object reference.

procedure division using aParent returning aBeepWindow.
* BeepWindow を作成し初期化するコード
 exit
```

```
method.

 end method "new".
```

BeepWindow は、実際にはクラスメソッドを定義せず、スーパークラスから "new" メソッドの実現方法を継承しています。ここでは、クラスメソッドを作成する方法を説明します。

クラスプログラム自身と同様に、メソッドのデータ部では各種のデータを宣言できます。data division 見出し自身は任意です。ここで宣言したデータは、このメソッド内のコードからだけアクセスできます。データ部では次の節が使用できます。

- working-storage
  メソッドが処理のために使用する変数。working-storage のデータは、メソッドを呼び出すたびに初期化し直されることはありません。さらにこのデータは、オブジェクトのすべてのインスタンスによって共有されるので、呼び出しのたびに異なるインスタンスによって上書きされないという保証もありません。

- local-storage
  メソッドによる再帰的作業をサポートするのに必要な変数。メソッドが再帰的に呼び出されると、再帰の各レベルごとに新しい local-storage データが作成されます。データ項目はメソッドコード内で初期化する必要があります。local-storage 内の value 句はコンパイラによって受け入れられますが、実行時には無効です。

- linkage
  プログラムとの間でパラメータとして受け渡しされる変数。

手続き部にはメソッドのためのコードが入っています。メソッドの処理は exit method 文によって終了します。この文は、メソッドを呼び出したプログラムに処理を返します。

5.1.7 オブジェクトプログラム

オブジェクトプログラムは、クラスのインスタンスのためのデータとメソッドを定義します。オブジェクトプログラムはクラスプログラム内に入れ子にされます。次のような構造になっています。

```
      object.

        object-storage section.

  * オブジェクト用インスタンスデータ

   ...

  * インスタンスメソッド

        end object.
```

オブジェクトプログラムの中で意味を持つデータ部の節は、object-storage 節だけです。他のデータ節も作成できますが、実行時にその節のデータをアクセスしようとした場合の動作は不確定です。

object-storage 節で宣言したデータは、すべてのインスタンス メソッドからアクセスでき、クラスのサブクラスのインスタンスによって継承できます。

オブジェクトプログラムには手続き部はありません。オブジェクトの初期化コードは、メソッドとして作成します。BeepWindow はメソッド "initialize" を使用します。Initialize は、BeepWindow のオブジェクトを作成するメソッド "new" によって呼び出されます。

テキスト エディタを end class-object 見出しのすぐ下までスクロールすると、オブジェクト プログラムとデータ宣言段落を表示できます。

5.1.8 インスタンスメソッド

インスタンス メソッドは、オブジェクト プログラムの入れ子のプログラムです。次のコードは、beepwin.cbl の "initialize" メソッドの概要です。

```
 method-id. "initialize".

      local-storage section.

...

 linkage section.

   01 lnkParent                  object reference.

procedure division using lnkParent.

...

 exit

method.

end method "initialize".
```

インスタンス メソッドは、オブジェクト プログラムの内側に入れ子にされます。インスタンス メソッドの作成はクラス メソッドの作成とほとんど同じです。違いは、インスタンス メソッドがアクセスできるデータの範囲だけです。

インスタンス メソッドは、次のデータをアクセスできます。

- インスタンスメソッドプログラムのデータ部で宣言されたデータ

- オブジェクトプログラムの object-storage 節で宣言されたデータ
- クラスプログラムのデータ部で宣言されたデータ。ただし、クラスプログラムの object-storage 節のデータは除きます。

BeepWindow インスタンス初期コードの確認

1. [ブラウザ] ウィンドウから "initialize" メソッドを選択して、テキスト エディタ ウィンドウに "initialize" メソッドを表示します。
2. 下方向へスクロールして、データ宣言とコードを確認します。このコードは、チュートリアルの後半で Animator を使用してアニメートします。

これでクラス構造の概要は終わりです。次は、デスクトップ上にウィンドウを作成するコードをアニメートします。

5.2 ウィンドウの作成

ここでは、BeepWindow を作成するコードを示して、デスクトップ上にこのウィンドウを表示します。 コードの動作を正確に確認できるようにするため、 Animator を使用してコードを作成していきます。 このセッションでは、提供されている NetExpress クラス ライブラリから 2 つのクラスを紹介します。

EventManager のインスタンスは、すべてのシステム イベントを処理し、各自のアプリケーションで必要なイベントを、それらのイベントを処理するオブジェクトに渡します。 これは、GUI オブジェクトの位置指定と調整システムも初期化します。

ウィンドウ BeepWindow はウィンドウのサブクラスであり、そのメソッドすべてを継承します。 BeepWindow がそれ自体に送信するメッセージの多くは、実際には Window（またはそのスーパークラスの 1 つ）によって実行されます。サブクラスのプログラマーは、メソッドがどこで実現されているかを知る必要はありません。メソッドがスーパークラスのパブリック インタフェースにあることだけを知っていればよいのです。

提供されているクラス ライブラリがデバッグ用にコンパイルされていないため、このクラスのコードは Animator では表示されません。

OO アプリケーションは、実行を開始するために短い手続きトリガ プログラムを必要とします。オブジェクトは、メッセージを受信するまで何もしないため、初期プログラムは、アプリケーションの動作を開始するために最初のメッセージを送信します。 アプリケーションが開始すると、その内部のオブジェクトは相互にメッセージを送信して、実行を続けます。

トリガ プログラムは、アプリケーションで使用するクラス ライブラリのロードも行います。基準クラス ライブラリは、RTS によって常に自動的にロードされますが、追加クラス ライブラリは CALL 文を使用してロードする必要があります。

チュートリアルのこの部分では、初期ウィンドウを示す図が表示されます。

BeepWindow を作成するコードの表示

1. デバッガを開始します。

   デバッガ ウィンドウは、強調表示されて、実行準備ができたトリガ プログラムのタグ T010 より下の行によって開きます。

2. 最初の文 (**call "apigui"**) をステップ実行します。

   GUI クラス ライブラリがロードされて、デスクトップ上にウィンドウを作成するために必要なグラフィック オブジェクト サポートすべてが提供されます。

3. 次の文 (**invoke EventManager...**) をステップ実行します。

   EventManager クラスのインスタンスが作成されます。このオブジェクトは、GUI イベントすべてを処理します。

4. 次の文 (**invoke anEventManager"initialize"**) をステップ実行します。

   EventManager インスタンスを初期化してから、GUI オブジェクトの作成を開始する必要があります。

5. 次の文 (**invoke anEventManager"getDesktop"...**) をステップ実行します。

   Windows デスクトップの表示へのハンドルが返されます。これは、BeepWindow の親として使用します。

6. タグ T030 (**invoke BeepWindow "new"...**) の後の行をステップ実行します。
   "new" メソッドは、Window から継承されます。Window のこのメソッドは、インスタンスを作成し、それに対して "initialize" メッセージを送信します。 BeepWindow は、"initialize" インスタンス メソッドを持つため、"new" がメッセージ "initialize" を送信すると、実行は BeepWindow の "initialize" メソッドに転送されます。 制御は、現在 beepwin.cbl (BeepWindow クラス) のタグ B011 の下にあります。

   パラメータ theDesktop も "initialize" メソッドに渡されます。これは、前のステップで EventManager から取得したオブジェクト ハンドルです。これは、Window インスタンスに親として渡されます。すべてのグラフィック オブジェクトに親が必要ですが、これは通常 Windows デスクトップか別のウィンドウです。グラフィック オブジェクトの親とそのスーパークラスを混同しないでください。オブジェクトは、その動作をスーパークラスから継承します。グラフィック オブジェクトの親は、実行時にこのオブジェクトを所有します。親が廃棄されると、その子オブジェクトもすべて廃棄されます。

7. タグ B011 の下の行 (**invoke super "initialize"...**) をステップ実行します。

   Window クラスから継承された "initialize" メソッドが上書きされました。しかし、Window "initialize" メソッドには、ウィンドウの作成の一部として実行が必要なコードが含まれます。

   invoke super "initialize" 文は、メッセージ "initialize" をこのオブジェクトに送信しますが、ランタイ

ム システムに対して、スーパークラス コード (Window クラスにある)でこのメソッドの検索を開始するように指示します。

8. タグ B012 の下の文 (**invoke self "setTitleZ...**) をステップ実行します。

   self にメッセージを送信すると、現在のオブジェクト内のメソッドが起動します。この場合は、BeepWindow クラスです。 Self は、それが検出されたオブジェクトを指すオブジェクト参照です。self を宣言する必要はありません。自動的に作成されます。

   "setTitleZ" メソッドは、スーパークラス Window から継承されたもう 1 つのメソッドです。

9. タグ B014 の下の行をステップ実行します。 これにより、パラメータが初期化されて、ウィンドウのサイズと位置が設定されます。サイズはピクセルで指定されます。

   文 **invoke self "setRectangle"...** は、ウィンドウのサイズと位置を設定します。これは、継承されたもう 1 つのメソッドです。

10. タグ B020 の下の文 (**invoke PushButton "new"...)** をステップ実行します。

    これにより、PushButton クラスのインスタンスが、self (BeepWindow のこのインスタンス) を親として作成されます。つまり、PushButton は、この BeepWindow と関連して位置設定されて、BeepWindow が表示可能になると表示されます。BeepWindow が廃棄されると、このインスタンスも廃棄されます。

11. タグ B021 の下の文をステップ実行して、ボタンにラベルを付けます。

12. タグ B023 の下の行をステップ実行します。

    これにより、親ウィンドウの左下端にボタンが位置設定されます。座標は常に、親の左上端との相対関係にあります。

13. ここで、プッシュボタンと BeepWindow に対してイベントを登録し、BeepWindow のエンドユーザがそれを制御できるようにします。 タグ B030 の下にある最初の文 (**map event p2ce-close...**) をステップ実行します。

    map 文は、Object COBOL の拡張機能であり、ボキャブラリ機能を使用して追加されます。チュートリアルの後半では、ボキャブラリについて学習します。

    この文は、イベントをメソッドにマッピングします。エンドユーザが BeepWindow の [システム] メニューで [閉じる] 項目をクリックすると、クローズ イベント (p2ce-close によって表わされ、コピーファイル p2cevent.cpy に定義される) が、オペレーティング システムによって出されます。

14. タグ B031 の下の文 (**map event p2ce-clicked...**) をステップ実行します。

    これは、ボタン オブジェクトでクリックされたイベントを self (ウィンドウ) の "beepButtonPushed" メソ

ッドにマッピングします。

15. **exit method** 文をステップ実行します。

これで、チュートリアルのこのセクションは終わりです。次のセクションはこのセクションから続けて説明するので、Animator を現在の状態のままにしておいてください。

次のセッションでは、ウィンドウに動作を追加して、ユーザ イベントを処理できるようにします。

5.3 イベントの処理

ここまでに、ウィンドウを作成してプッシュボタンを追加し、ウィンドウ コントロールから BeepWindow クラス内のメソッドにイベントをマップしました。 ここでは、イベントの処理を開始して、BeepWindow に動作を開始させます。

GUI アプリケーションのエンドユーザが何か（たとえば、マウスの移動、キーを押す、ボタンを押すなど）を実行するたびに、オペレーティング システムはアプリケーションにイベントを送信します。 イベントは、アプリケーションに対して、何かが発生したことを通知します。

イベント処理は、EventManager クラスのインスタンスによって処理されます。EventManager に "run" メッセージを送信すると、イベントがポーリングされます。 マップしたイベントのどれかが発生すると、EventManager はそのイベントにマップされたメソッドを起動します。

EventManager イベントのポーリングは、"terminate" メッセージを送信することによって停止します。 これにより、アプリケーションによって作成されたすべての GUI オブジェクトも閉じます。イベント登録コードをステップごとに進めて、イベントに応答する BeepWindow を監視することによって、この動作を確認できます。

イベント処理のアニメート

このセクションは、BeepWindow を作成したコードを表示した前のセクションからの続きです。Animator 実行点は、タグ T040 の上の文 (**invoke aWindow "show"**) を強調表示していなければなりません。

1. 行 **invoke aWindow "show"** をステップ実行します。

   BeepWindow は、デスクトップ上に表示されます (これを表示するためには、NetExpress 統合開発環境を最小化しなければならない場合があります)。 ただし、ボタンは表示されず、ボタンの上にマウス カーソルを移動すると、使用中の状態 (通常は砂時計)に変わります。

   アプリケーションはまだイベントを処理していないため、BeepWindow は何も応答しません。

2. タグ T040 の下の文 (**invoke anEventManager "run"**) をステップ実行します。

   制御は EventManager に渡されて、オペレーティング システムからのイベントを待機しています。
   EventManager は、デバッグ用にコンパイルされていないため、制御が BeepWindow に戻るまで、何も表示

されません。

3. Beeper ウィンドウの [Beep] プッシュボタンを押します (Beeper ウィンドウは、NetExpress ウィンドウの後ろに消えている場合があります)。

   これにより、"beepButtonPushed" メソッドにマッピングされているプッシュボタンでクリックされたイベントが発生します。 実行点は、メソッド "beepButtonPushed" 内にあるタグ B050 の下の行 (**invoke alarm "warning"**) に変わります。

4. **invoke** 文をステップ実行します。

   これにより、システム ベルが鳴ります。アラームは、システム ベルを制御するクラス ライブラリのクラスです。 1 つのベルしかないため、Alarm のインスタンスを作成することはできません。すべてのメソッドがクラスそのものによって実現されます。

5. **exit method** 文をステップ実行します。

   制御は EventManager に戻ります。

6. BeepWindow ウィンドウで、[システム] メニューの [閉じる] を選択します。

   これにより、BeepWindow の"closeWindow" メソッドにマッピングされている、ウィンドウのクローズ イベントが発生します。 実行点は、このメソッドの最初の行に変わります。

7. 最初の文 (**invoke EventManager "new"...**) をステップ実行します。

   EventManager の "new" メソッドは、ほとんどのクラスの "new" メソッドよりも少し違って動作します。アプリケーションには 1 つの EventManager しか存在できないため、"new" をクラスに送信すると、そのたびに新しいクラスを作成するのではなく、常に同じオブジェクトが返されます。

   この文は、BeepWindow に、このアプリケーションの EventManager オブジェクト処理イベントに対するハンドルを与えます。

8. タグ B040 の下の文 (**invoke lsEventManager "terminate"**) をステップ実行します。

   これは、EventManager に対して、イベント処理を停止するよう指示します。また、アプリケーション内のすべての GUI オブジェクトを廃棄するため、Beep ウィンドウはデスクトップから消えます。

9. **exit method** 文をステップ実行します。

   制御はトリガ プログラムに戻ります。

10. [アニメート] メニューの [アニメート停止] をクリックします。

5.4 まとめ

これでクラスプログラムの作成に関するチュートリアルが終了しました。ここでは次のことを学習しました。

- クラスプログラムを作成する方法
- GUIオブジェクトを作成する方法
- 簡単な GUI イベントの処理

この後のチュートリアルでは、ビジネスロジックからインタフェースを細かく分離できる、より複雑なイベント処理について学習します。

第6章　継承のチュートリアル

このチュートリアルでは、Object COBOL で継承がどのように動作するかを見ていきます。ここでは、各種の銀行口座の例を使用し、1 つのクラスから共通の動作を継承し、個々の口座について新しい動作を追加したり、既存の動作を変更する方法を示します。 このチュートリアルは、次のセッションからできています。

1. Account クラス
2. 簡単な口座トランザクション
3. まとめ

学習時間：25分

6.1 Account クラス

ここでは、このチュートリアルで使用する Account クラスを紹介します。まず、NetExpress 統合開発環境を使ってこのクラスにあるコードを表示してから、次のセッションでアニメートします。 全部で 4 つの account クラスがあります。

Account	抽象クラス
CheckAccount	Account のサブクラス。当座預金口座のための動作を追加する (払い戻しに対する当座貸越高のチェック)。
SavingsAccount	Account のサブクラス。普通預金口座のための動作を追加する (利子の支払い)。
HighRateAccount	SavingsAccount のサブクラス。口座開設時の最小残高チェックを追加する。

account クラス コードの表示方法

1. NetExpress 統合開発環境を開始して、継承プロジェクト (bank¥inherit.app) をロードします。
2. [プロジェクト] メニューの [リビルド] をクリックして、プロジェクト内のファイルすべてをリビルドします。
3. [検索] メニューの [ブラウザ] をクリックして、[ブラウザ] ウィンドウを表示します。
4. base の隣にあるプラス (+) 記号をクリックして、Account サブクラスを確認します。

   Account には、CheckAccount および SavingsAccount の 2 つのサブクラスがあります。SavingsAccount には、さらに、[ブラウザ] ウィンドウのプラス記号 (+) で示されるサブクラスがあります。
5. savingsaccount の隣にあるプラス (+) 記号をクリックして、そのサブクラスを確認します。

ここで、4 つの型の account クラス、Account、 CheckAccount、SavingsAccount、および HighRateAccount があることがわかります。階層ビューは、Account クラスが、他のクラスすべてのスーパークラスであることを示しています。Account は、異なる型の bank account すべてに共通な動作を実現する抽象クラスです。

6. [ブラウザ] ペインの一番上にあるボタンをクリックして、[ブラウザ] ウィンドウにメソッドだけを表示し、[メソッド] ボタンだけを押されている状態で示します。

7. account を選択して、account クラスによって実現されたメソッドを確認します。

   Account には、"openAccount" および "setNextAccountNumber" の 2 つのクラス メソッドと、いくつかのインスタンス メソッドがあります。 これらには、"deposit" や "withdraw" のように、bank account で実行できると期待されるすべての機能のメソッドが含まれます。

8. savingsaccount を選択します。

   SavingsAccount クラスが実現するメソッドは、Account よりもはるかに少なくなります。これには、口座開設、預け入れ、または払い戻しのためのメソッドがありません。このクラスは、これらのメソッドを示すコードすべてを Account から継承します。 このクラスには、利子を加算するためのメソッドがありますが、これはすべての型の account に共通する動作ではないためです。また、"printStatement" の実現もあります。すべての account クラスが、"printStatement" の独自の実現を持ちます。これは、異なる型の account がすべて、異なる型の文を印刷するためです。これは、多相性の一例です。

これで、このチュートリアルにおける次の作業、Bank1 のアニメートを行なう準備ができました。

6.2 簡単な口座トランザクション

ここでは、簡単な口座トランザクションと継承が実際どのように動作するかを見るために、bank1.cbl をアニメートします。次の手順は、このチュートリアルの前半からの続きです。したがって、NetExpress 統合開発環境が起動され、account クラスが実行されていることを仮定しています。

bank1のアニメート

1. [アニメート] メニューの [アニメート開始］をクリックして、Animator を起動します。[アニメート開始] ダイアログ ボックスが表示されたら、[OK] ボタンをクリックします。

   bank1.int が実行用にロードされます。これで、Bank1 が表示されます。最初の文のグループ (タグ B001 の下) は、カスタマ用の新しい CheckAccount を作成します。

2. 最初の 3 つの文をステップ実行します。

   invoke CheckAccount をステップ実行すると、Account クラスのタグ A001 の下にある行に実行が切り替

えられます。CheckAccount は、"openAccount" メソッドを実現しませんが、これを Account から継承します。

3. 文 **invoke super "new"** をステップ実行します。

   "new" メッセージがこのオブジェクトに送信されます。しかし、ランタイムシステムには、Account のスーパークラス (クラス Base) の用のコード内でメソッドの検索を開始するように指示されます。ランタイムシステムは、CheckAccount のインスタンスを返します。

4. タグ A002 の下にある **invoke** 文をステップ実行します。

   Account の "setNextAccountNumber" メソッドに移動します。

5. このメソッドの最初の文（**if nextAccountNumber = 0**）をステップ実行します。

   データ項目 nextAccountNumber は、0 の value 句によって宣言されています。account クラス、またはそのサブクラスのどれかが最初にロードされる時に、このデータ項目は 0 に設定されます。メソッド "setNextAccountNumber" はこの値をテストし、ゼロの場合、初期値を設定します。CheckAccount クラスが "openAccount" メッセージを受け取るのはこれが初めてなので、nextAccountNumber を初期化する必要があります。

6. 次の文 (**invoke ExceptionManager...**) をステップ実行します。.

   この文では、account クラスの例外メッセージ ファイルが設定されます (関連のチュートリアルを参照)。

7. 次の文（**invoke self "getFirstAccountNumber"**）をステップ実行します。

   実行は、CheckAccount の "getFirstAccountNumber" メソッドに切り替えられます。この場合、self は CheckAccount クラス オブジェクトを参照します。Account クラスによって実現されたコードを実行していましたが、元の "openAccount" メッセージは CheckAccount クラス オブジェクトに送信されています。CheckAccount クラス オブジェクトは、Account から "openAccount" メソッドを 継承しています。このメソッドは、口座の番号付けを始めるために "openAccount" メソッドのための値を返します。Account の各サブクラスは異なる値から口座番号を開始します。すべてのサブクラスは開始値を返すために "getFirstAccountNumber" を実現しています。

8. **exit method** 文までコードをステップ実行します。

   "setNextAccountNumber" の **exit method** 文に戻ります。

9. **exit method** 文をステップ実行します。

   "openAccount" メソッドに戻ります。

10. 次の文（**invoke lsAccount "initAccount" ...**）をステップ実行します。

    新しく作成された account オブジェクトに対するオブジェクト ハンドルは、lsAccount に入っています。"initAccount" メソッドは、account 属性を、名前、残高、および口座番号によって設定します。

11. **exit method** までコードをステップ実行します。

    "openAccount" の **exit method** 文に戻ります。

12. **exit method** 文をステップ実行します。

    実行は、CheckAccount クラスへのオブジェクト ハンドルを持つようになった Bank1 に戻ります。

13. タグ B002 の下にある、2 つの文をステップ実行します。

    CheckAccount インスタンスに $1000、預金されます。このコードも、すべて Account クラスにあります。

14. Bank1 の タグ B003 にある文に戻るまで、Account コードをステップ実行します。

15. タグ B003 の下にある、2 つの文をステップ実行します。

    CheckAccount インスタンスから $50、引き出されます。ここで、invoke が実行を CheckAccount 内のコードに切り替えます。これは、CheckAccount クラスが、独自の "withdraw" メソッドを実現しているからです。このメソッドは、引き出し金額が当座貸越しの許容金額を超えていないかをチェックします。

16. タグ C010 の下にある文をステップ実行します。

    引き出し金額が、CheckAccount のこのインスタンスについて設定された当座貸し越し高を超えると、オブジェクトは例外 (エラー条件) を発生します。 ここでは引き出し金額は OK であるため、実行は通常どおりに進められます。この場合、アプリケーションには、account の例外を取り扱うための例外ハンドラが登録されていないので、エラーはシステムレベルでトラップされ、エラー番号が表示され、アプリケーションの実行が停止します。他の NetExpress チュートリアルで使用している BankApplication の例でも、これらの account クラスを使用しますが、この種のエラーに対処するための例外ハンドラが登録されています。

17. タグ B007 に達するまでアプリケーションをステップ実行します。

    Bank1 が以前に使用していない型の account を開くたびに、初期口座番号を取得するためにコードをステップ実行する必要があります。コードを実行すると、異なる型の銀行口座のほとんどのコードが、Account クラスから継承されており、一部のメソッドだけがサブクラスで実現されていることがわかります。サブクラスはすべて、データとともに Account クラスを継承します (各プログラムの最初にある **class-id** 文を参照)。つまり、サブクラスは Account のメソッドを継承するだけでなく、account で宣言されたデータ項目への直接アクセス権も持ちます。異なる account 型はすべて、変数 balance、aName、および accountNumber

にアクセスできます。**with data** 句が Account サブクラスの class-id にないと、このデータに直接アクセスできません。代わりに、このデータにアクセスするには、スーパークラスにメッセージ (**invoke super...**) を送る必要があります。Account クラスのプログラマは、これらのメッセージに応答するためのメソッドを実現する必要があります。

18. B007 の下の文をステップ実行して、HighRateAccount を開きます。

    HighRateAccount のタグ H010 に進みます。HighRateAccount は、口座開設に使われた現金額をチェックするために、独自の "openAccount" を実現しています。 しかし、一度これが検査されても、メッセージを super に送信することによって、Account の "openAccount" メソッドが引き続き使用されます。

19. Bank1 のタグ B008 に達するまで、残りのコードをステップ実行します。

    このコードは、共通クラスの子孫であるオブジェクト間の多相性の使用を示すものです。 すべての account 型は、それぞれに適切な文を印刷するために "printStatement" メソッドを実現しています。

20. 文を印刷するためのコードをステップ実行します。

6.3 まとめ

これで継承のチュートリアルが終了しました。ここでは、次のことを学習しました。

- 継承を使ってコードを再利用する方法
- 多相性の使い方

第7章　コレクション、固有値、ディクショナリのチュートリアル

このチュートリアルでは、コレクション クラスの使い方について学習します。コレクションは、オブジェクト指向プログラミングでは非常に便利な概念です。コレクションは、オブジェクトのグループを格納するのに使います。通常は同じ種類のオブジェクトを 1 つのコレクションに格納しますが、異なる種類のオブジェクトを同じコレクションに格納することもできます。

Object COBOL では、COBOL の固有データを格納することもできます。固有データはコレクション中ではオブジェクトによって表現されません。固有データの格納には、固有データの項目（例えば、PIC X(20)）をオブジェクトのようにして取り扱う機構を使用します。固有データとオブジェクトを 1 つのコレクション内で混合したり、異なる種類の固有データを混合することはできません。

このチュートリアルの最後ではディクショナリを説明します。ディクショナリもコレクションの一種です。

このチュートリアルは次のセッションからできています。

1. コレクションの分類
2. 固有値の使い方
3. アソシエートとディクショナリ
4. 繰返しメソッド
5. まとめ

学習時間：25分

7.1 コレクションの分類

クラス ライブラリの各コレクションは、次の属性によって分類できます。

- 索引付きまたは索引なし

  索引付きコレクションの各要素は、その位置を与えることによってアクセスできます。これは配列を使用する場合と同じです。索引なしコレクションでは、要素を格納するために決められた順序はありません。

- 自動成長または手動成長

  自動成長のコレクションは、コレクションを作成したときの容量を超えると大きくなります。手動成長のコレクションは、メッセージ "grow" が送られたときだけ大きくなります。

- 重複許可または重複禁止

  コレクションによっては項目の重複が禁止されているときがあります。この場合、コレクションにすでにあるのと同じ値の項目を追加しようとすると、例外が起こります。

コレクション クラスとその属性の一覧を次に示します。

Bag	索引なし、自動成長、重複許可
Array	索引付き、手動成長、重複許可
CharacterArray	索引付き、手動成長、重複許可
OrderedCollection	挿入順による索引付き、自動成長、重複許可
SortedCollection	ソート順による索引付き、自動成長、重複許可
ValueSet	索引なし、自動成長、重複値禁止

次に coll0.cbl という簡単な COBOL プログラムを見ます。このプログラムは、おもなコレクションの類似点と相違点を示しています。このプログラムは Object COBOL のクラスプログラムではなく、クラス ライブラリのコレクションオブジェクトを使用する手続き COBOL コードです。

coll0.cblの実行

1. NetExpress 統合開発環境を開始して、Objects プロジェクト (collect0¥coll0.app) をロードします。
2. プロジェクトをリビルドします。
3. Animator を開始します。
4. コードの実行を開始する前に、fruitData の宣言が表示されるまで、Working-Storage 節をスクロールします。これは、プログラムが異なる型のコレクションを格納するための文字列セットです。
5. 実行点にカーソルを返します。
6. 最初の文 A001 (**invoke Array "ofReferences"...**) をステップ実行します。

   オブジェクトを格納するための配列が作成されます。任意の型のコレクションを、参照のコレクションまたは値のコレクションとして、CharacterArray とは別に作成できます。 参照のコレクションにはオブジェクトが格納されます。値のコレクションには COBOL 固有データが格納されます。これについては、このチュートリアルで説明します。

7. A001 ～ A005 の文をステップ実行して、異なる型のコレクションのインスタンスを作成します。

   すべての型のコレクションに、初期サイズを指定する必要があります。Array の場合は、コレクションに

"grow" メッセージを送信しないかぎり、その初期容量を超えることはできません。他の型のコレクションは、最初に指定された以上の要素を追加すると、サイズが大きくなります。 ただし、コレクションの成長は実行時に大きな処理負担となります。したがって、引き続き、常に初期サイズを指定するようにして、コレクションが成長する回数を減らす必要があります。

8. タグ A006 の下にある文 (**move 20 to i**) をステップ実行します。異なる型のコレクションで要素として作成される CharacterArray のインスタンスの長さが、データ項目 i に入ります。

9. タグ A007 の下にある文 (**invoke CharacterArray "withLengthValue"...**) をステップ実行します。

   CharacterArray は、文字列を格納するためのオブジェクトです。メッセージ "withLengthValue" は、CharacterArray の新しいインスタンスを作成し、データによって初期化します。

10. タグ A008 (**invoke aBag "add"...**) の下にある文を **end-perform** 文までステップ実行します。

    Array を除くすべての異なるコレクション 型では、メッセージ "add" を使用して新しい要素を追加します。Array は自動的に成長しないので、指定された索引位置に要素を格納する "atPut" を使用します。OrderedCollection は索引付きですが、コレクションに要素を追加しない限り、"atPut" を使用できません。

    たとえば、"add" を使用して最初の 5 つの要素を追加した場合は、"atPut" と 1 ～ 5 の間の索引を使用して、これらの要素を置換できます。 5 を超える索引を使用して、"atPut" を実行するには、さらに要素を追加する必要があります。SortedCollection に要素を挿入するために "atPut" を使用することはできません。SortedCollection は、各要素の値を使用して、コレクション内でのその位置を判断します。

11. テキスト カーソルをタグ A010 の下にある文 (**move 4 to loopCount**) に移動して、[アニメート] メニューから [カーソル位置まで実行] を選択します。これで、各文を順番に実行しなくても、PERFORM ループの実行を完了できます。

12. タグ A011 (**display " "**) まで文をステップ実行します。

    Array インスタンス内の 4 番めの要素に対するオブジェクト参照が抽出されます。"display" メッセージを CharacterArray に送信すると、これは画面上に表示されます。この内容は、"banana" という語でなければなりません。

13. 次の文 (**invoke aBag "includes"...**) をステップ実行します。

    Bag は索引が付いていないので、直接、要素を抽出することはできません。そのかわり指定した値のオブジェクトが（ 1 つまたは複数）あるかを問い合わせることはできます。メッセージ "includes" は、一致する値のオブジェクトが Bag にある場合は 1、ない場合は 0 を返します。繰返し メソッドを使用すると、索引がついていないコレクション (Bags and ValueSets) の内容を検討することもできます。

14. タグ A012 (**display " "**) まで文をステップ実行します。

これらの文は戻り値の結果をテストし、一致する値のオブジェクトが bag にあるかどうかを通知します。

15. タグ A012 の下にある文 (**invoke aBag "add"...**) をステップ実行します。これによって、文字列が再び Bag に追加されます。異なる値を持つ各オブジェクトのオカレンス数を記録することによって、Bag に重複する値を持つオブジェクトが格納されます。

16. タグ A013 の下にある文 (**invoke aBag "occurrencesOf"...**) をステップ実行します。

    メッセージ "occurrencesOf" は、指定されたオブジェクトと値が一致するコレクションの要素の数を返します。

17. タグ A014 (**display " "**) まで文をステップ実行します。

    Bag には指定された文字列のオカレンスが 2 つあることが表示されます。

18. タグ A014 (**invoke anArray "occurrencesOf"...**) と A015 (**display " "**) の間の文をステップ実行します。

    このコードは、Array がメッセージ "occurrencesOf" に対して応答する方法を示しています。 索引付きコレクションには、Bag や ValueSet のインスタンスのほかに、"occurrencesOf" と "includes" が使用できます。これは、要素の位置ではなく要素の有無を示します。

19. タグ A015 の下にある最初の文 (**invoke aValueSet "add"...**) をステップ実行します。

    ValueSet のインスタンスは、重複する要素を持ちません。ValueSet には、すでに "banana" という値の要素が入っており、同じ値の要素が再び追加されることはありません。

20. 次の文 (**invoke aValueSet "occurrencesOf"...**) をステップ実行します。

    このメッセージは、ValueSet のインスタンスについて、常に 1 または 0 を返します。

21. タグ A016 (**display " "**) まで残りの文をステップ実行します。

    メッセージ "occurrencesOf" の結果が表示されます。

22. [アニメート] メニューの [実行] を選択して、プログラム内の残りのコードを実行します。

    タグ A016 (**display "Collection contents"**) の下にある文は、OrderedCollection と SortedCollection の文字列すべてを索引順に表示します。OrderedCollection の要素は、追加された順に表示されます。
    SortedCollection の要素は、アルファベットの昇順でソートされて表示されます。SortedCollection のデフォルトは昇順です。

これで、チュートリアルのこのセッションは終わりです。コードの検討を終えたら、Animator ウィンドウを閉じる

ことができます。

このセッションを終えると、次のような疑問が生ずるかもしれません。

- コレクションオブジェクトはどのようにして要素の値を見つけるのか ？
  すべての要素がオブジェクトの場合、オブジェクトのカプセル化が壊れることはないか ？

- 同様に SortedCollection は、ある要素の値が他の要素より大きいかを、どのようにして判断するのか ？

プログラム coll0.cbl は、CharacterArray のインスタンスを、それが作成したコレクションに格納します。CharacterArray のインスタンスは、明らかな 1 つの値（ユーザが格納した値）を持つ単純なオブジェクトです。しかし、継承のチュートリアルで使ったような Account オブジェクトを格納した場合、どうやって値を判断するのでしょうか。

コレクションオブジェクトには、その中に要素として格納されたオブジェクトが動作しなければならないフレームワークが用意されています。2 つのオブジェクトが等しいかどうかをコレクションが知りたい場合、一方のオブジェクトにメッセージ "equal" を送り、もう一方のオブジェクトには "equal" をパラメータとして送ります。この後、オブジェクトはもう 1 つのオブジェクトを照会し、両者が等しいかどうかを判断します。

SortedCollection のデフォルトのソート方法は、これと同じ動作をします。SortedCollection はある要素にメソッド "lessThanOrEqual" を送り、もう 1 つの要素には "lessThanOrEqual" をパラメータとして送ります。受け取った要素は、自分自身をもう 1 つの要素と比較し、結果を返します。

独自のオブジェクトを作成し、コレクションに格納する場合、これらのメソッドを自分で実現する必要があります。ただし、CharacterArray などのクラスからサブクラスを作る場合、メソッドは自動的に実現されるので自分で用意する必要はありません。またBaseには、2 つのオブジェクトのオブジェクトハンドルを比較するデフォルトのメソッド "equal" があります。この "equal" は、2 つの要素が実際に同じオブジェクトである場合だけ等しいと判断します。

7.2 固有値の使い方

前のセッションでは、各型のコレクションにオブジェクトを格納するためのプログラムを見ました。しかし、数値などの固有 COBOL データをコレクションに格納したい場合もあります。これには、クラス ライブラリの固有クラスを使用します。

Object COBOL には、固有データ項目に対し、INVOKE ... AS 文を使用して、オブジェクトと同様にメッセージを送信するための機構があります。

メッセージを固有データに送信するには、まず固有クラスに、使用している固有データの型と長さに合わせて新しいクラスを作成させる必要があります。 クラス ライブラリには、3 つの異なる型の固有データ (PIC X、PIC X COMP-X、PIC X COMP-5) 用のクラスがあります。これらのクラスは、固定長データを扱うテンプレートです。クラスをクローン化する場合は、処理するデータの実際のサイズを指定します。

次に、サンプル プログラム coll1.cbl を示します。このプログラムは固有クラスを使用して、1 組の整数を配列に格納します。

coll1.cbl の実行

1. NetExpress 統合開発環境を開始して、プロジェクト (collect1¥coll1.app) をロードします。
2. プロジェクトをリビルドします。
3. Animator を開始します。
4. タグ A001 (**move 4 to i**) の下にある 2 つの文をステップ実行します。

   "newClass" メッセージは、CobolComp5 クラスのクローンを作成します。これはこの場合、4 バイトの長さのデータに合わせて初期化されています。 返されるオブジェクトは、CobolComp5 のインスタンスではなく、新しいクラス オブジェクトです。
5. タグ A002 (**move 10 to i**) の下にある 2 つの文をステップ実行します。

   10 個の要素のためのスペースを持つ Array のインスタンスが作成されます。今回、Array インスタンスを作成するためのメッセージは "ofValues" です (前のセッションの例では、"ofReferences" でした)。"ofValues" メッセージによってコレクションが作成されると、オブジェクト ハンドラの代わりに固有データを格納します。

   クローン化されたクラス PicX4Comp5 は、テンプレートとして使用されるため、Array は各要素に割り当てる容量を知っています。 単一のコレクション内でオブジェクトと固有データを混在させることはできません。コレクションを作成したら、初期化対象となるデータの型だけを格納できます。

   複数の異なる種類のデータを 1 つのコレクションに混在させる場合は、"ofReferences" を使用してコレクションを作成し、異なる型のオブジェクトを使用して、異なる型のデータを表示する必要があります。参照のコレクション内では、異なる種類のオブジェクトの混在に関する制約はありません。
6. タグ A003 (**move 10 to element**) の下にある最初の文をステップ実行します。
7. 実行点が次の文 (**perform varying from 1 ...**) にある間、実行ループを実行します。

   配列を初期化するための、PERFORM ループ全体が実行されます。
8. タグ A004 の下にある文をステップ実行します。

   配列から 4 番めの要素 (7 の値を持つ) 抽出され、表示されます。データ項目が返されるので、**display** 文を使用してこのデータ項目を表示できます。前のチュートリアルで説明した coll0.cbl のアニメート中にオブジェクトが返された場合は、それらのオブジェクトに "display" メッセージを送信して、表示する必要があります。

これで、チュートリアルのこのセッションは終わりです。Animator を閉じてください。

7.3 アソシエートとディクショナリ

ディクショナリは、キーとデータのペア（アソシエートと呼ぶ）を格納するための特殊な索引付きコレクションです。ディクショナリでは、データを格納または抽出するときの索引としてキーを使用します。ディクショナリでは、重複キーを格納することはできません。他のコレクション型と同様に、ディクショナリにはオブジェクトまたは固有データを格納できます。しかし、ディクショナリでは、キー部分とデータ部分がそれぞれ固有値にもオブジェクトにもなることができます。

ディクショナリを作成する場合、キー部分とデータ部分をどのように格納するかを指定するために、テンプレートを与える必要があります。テンプレートはAssociationクラスか、Association クラスのクローンです。

Association クラスは、データ格納のためのテンプレートの作成に使われる固有データクラスのためのクラスと同様にクローン化可能なクラスです。Association テンプレートは、実際には 2 つのテンプレートからできています。1 つはキー用、もう 1 つはデータ用です。どんな型のディクショナリを作成する場合も、Association テンプレートを作成する必要があります。

ディクショナリ オブジェクトのためのテンプレートを作成したら、ディクショナリ自身を作成するには、"ofValues" と "ofAssociations" の 2 種類の手段があります。ディクショナリ "ofValues" は各要素をキーとデータのペアとして格納します。ディクショナリ "ofAssociations" は、各要素をディクショナリの作成に使った Association テンプレートのインスタンスとして格納します。

ここでも Bank アプリケーションを紹介します。このアプリケーションは、Account オブジェクトの作成と作業のための GUI フロントエンドを準備します。Bank アプリケーションは、Object COBOL によるオブジェクト指向プログラミングのさまざまな特徴のデモンストレーションで、後続のチュートリアルでも使用します。

ここでは、Bank アプリケーションを使ってディクショナリの使い方を見て行きます。作成した口座は、口座番号をキーとしてディクショナリに格納されます。次に、AccountManager クラスのコード (accmgrd.cbl) を表示します。このコードは、ディクショナリ オブジェクトを使用して、アカウントを格納して抽出します。

ディクショナリ作成コードのアニメート

1. NetExpress 統合開発環境を開始して、プロジェクト (bank¥resbank.app) をロードします。
2. プロジェクト resbank.app をリビルドします。

   次に、AccountManager クラス内の実行するメソッドを検索し、これらのメソッドにブレークポイントを設定します。
3. [ブラウザ] ウィンドウを開きます。
4. AccountManager クラスを選択します (これは、Base のサブクラスであるため、base の下のエントリを拡張する必要があります)。

5. [ブラウザ]ウィンドウの [initialize] をクリックして、"initialize" メソッドをテキスト エディタのウィ ンドウにロードします。

6. 該当する行で右ボタンをクリックして、タグ M002 の下の文(**set nilReference to null**)にブレークポイントを設定してから、ポップアップ メニューの [ブレークポイント設定] をクリックします。

7. [ブラウザ]ウィンドウの [add] をクリックして、"add" メソッドに移動します。

8. このメソッドの最初の文 (**if lnkAccount <> null**) にブレークポイントを設定します。

9. "getAccount" メソッドに移動します。

10. このメソッドの最初の文 (**invoke accountCollection "at"...**) にブレークポイントを設定します。

11. [アニメート] メニューの [実行] をクリックするか、またはツールバーの [実行] アイコンをクリックして、アプリケーションを実行します。

    Animator は、一時停止した後、AccountManager の "initialize" メソッド内で停止します。このメソッドは、キーとして **pic x(8)** 固有データ、データとしてオブジェクトを持つディクショナリのインスタンスを作成します。

12. 最初の文 (**set nilReference to null**) をステップ実行します。

    オブジェクト参照データ項目がヌルに設定されます。この項目は、オブジェクトを保持するデータ テンプレート (ヌル テンプレート)を設定するために使用されます。

13. 次の 2 つの文 (**move 8 to i**) をステップ実行します。

    CobolPicX クラスがクローン化されて、8 バイト長の文字列用テンプレートが作成されます。これは、キー テンプレートです。

14. タグ M003 の下の文 (**invoke Association "newClass"...**) をステップ実行します。

    これで、ディクショナリに格納されるキーとデータの組み合わせ用の Association テンプレートが作成されました。

15. タグ M004 の下の文 (**invoke Dictionary "ofAssociations"...**) をステップ実行します。

    AccountManager が account データを格納するために使用するディクショナリが作成されます。

16. [アニメート] メニューの [実行] をクリックするか、またはツールバーの [実行] アイコンをクリックして、プログラムを実行します。

    このメソッドの残った部分は、別のディクショナリを作成して、そのディクショナリの ExceptionHandler を

設定します。例外処理については、この後のチュートリアルで説明します。

17. [銀行] ウィンドウのツールバーにある最初のボタンをクリックして、Fred の当座預金口座を残高 $10.00 で開設します。まず、[預金口座の開設] ダイアログ ボックスに、口座の詳細を入力します。作業が終了したら、[OK] ボタンをクリックします。

18. 最初の文 (**if lnkAccount <> null**) をステップ実行します。

    メソッドにオブジェクト ハンドルが渡されたかどうかだけがチェックされます。

19. [アニメート] メニューの [指定範囲を実行] をクリックするか、またはツールバーの [実行(現行のみ)] アイコンをクリックして、次の文 (**invoke lnkAccount "getAccountNumber"...**) をステップ実行します。

20. 次の文 (**invoke accountCollection "atPut" using...**) をステップ実行します。

    口座番号をキーとして、ディクショナリに口座が格納されます。

21. プログラムを実行します。

    表示されるメッセージ ボックスで [OK] ボタンをクリックします。

22. [Bank] ウィンドウの 3 番めのボタンをクリックして、口座 10000000 (作成したばかりのもの) を抽出します。さらに、[口座番号検索] ダイアログ ボックスの [OK] ボタンをクリックします (この [預金口座番号] フィールドには、すでに 10000000 が入力されています)。

    Animator は、AccountManager インスタンス メソッド "getAccount" の内部で停止します。

23. このメソッドの最初の文 (**invoke accountCollection "at"...**) をステップ実行します。

    ディクショナリから口座が抽出されて、参照が anAccount に挿入されます。

24. プログラムを実行します。

    BankApplication は、[預金口座明細] ダイアログ ボックスを開きます。このダイアログ ボックスには、account オブジェクトに保持された情報が表示されます。

25. [Cancel] ボタンを押して、このダイアログ ボックスを終了します。

26. NetExpress IDE から、[アニメート] メニューの [ブレークポイント] をクリックします。サブメニューの [プログラム内のすべてをクリア] をクリックします。

これで、このセクションは終わりです。次のセクションはこのセクションに続けて説明するため、Animator をシャット ダウンしないでください。次は、実行中のBankアプリケーションを使って、コレクションの繰返しメソッドを見ていきます。

7.4 繰返しメソッド

コレクション クラスには、単一のメッセージを使用してコレクションの要素すべてを検討できる、繰返しメソッドがあります。ここでは、前のセクションに引き続き、繰返しを使用して、Bank アプリケーションによって使用されるディクショナリを検索する方法を説明します。

ディクショナリ繰返しコードのアニメート

1. AccountManager の "retrieve" メソッドの最初の文にブレークポイントを設定します。
2. Wilma の普通預金口座を開きます。
3. [銀行] ウィンドウの 2 番めのボタンを押して、[口座名義で検索] ダイアログ ボックスを開きます。

   この機能では、一致する名前を見つけるために口座が検索されます。
4. [名前] フィールドに "Wilma" と入力して、[OK] ボタンを押します。

   実行点は、AccountManager のメソッド "retrieve" の最初の文に切り替わります。
5. タグ M050 (**`invoke CallBack "new"...`**) の下にある最初の文に達するまでステップ実行します。

   検索対象の名前を含む文字列オブジェクトが作成されます。
6. CallBack を作成する文をステップ実行します。

   nameString がパラメータとしてメッセージ "new" に渡されて、CallBack が作成されます。CallBack のメソッド ("searchAccounts") が呼び出されるたびに、"invoke" メッセージとともにのパラメータより先に nameString が渡されます。
7. タグ M051 (**`invoke accountCollection "select"...`**) の下にある文をステップ実行します。

   実行は、AccountManager "searchAccounts" メソッドの最初の文にジャンプします。この文には、検索中の名前を持つ文字列オブジェクトと、コレクション内の最初の口座が渡されています。
8. 文 **`if acName = srchName`** までアプリケーションを実行します。

   この時点で、acName と srchName には、口座からの名前と検索中の名前が含まれます。どちらの名前も、大文字小文字の不一致を避けるために、大文字に設定されています。[acName] をダブルクリックすると、"FRED" が含まれていることを確認できます。
9. 次の文をステップ実行します。

テストが失敗します。

10. **exit method** 文までステップ実行します。

    レベル 88 の項目 noMatch が真に設定されます。このメソッドを終了すると、反復を実行しているディクショナリのメソッド "select" 内のコードに 0 が返されます。

11. **exit method** 文をステップ実行します。

    実行はメソッド "searchAccounts" の最初に戻り、コレクションはこのメソッドを次の口座と共に呼び出します。

12. メソッドのコードをステップ実行します。

    今回、名前は Wilma の口座であるため一致します。メソッドは、ディクショナリに 1 の値を返し、作成される結果のコレクションの要素として Wilma が追加されます。

    exit method 文をステップ実行すると、実行はタグ M052 の下にある文に戻ります。"select" の結果は新しいディクショナリです。これには真とマークされた (CallBack の選択によって値 1 が返された)要素だけが入っています。

13. 次の文 (**invoke resultDictonary asOrderedCollection...**) をステップ実行します。

    メソッド "select" とメソッド "reject" は、元のコレクションと同じ型のコレクションを常に返します。この場合は、ディクショナリです。このメッセージは、ディクショナリのデータ部分を OrderedCollection にコピーします。 このメソッドにある他の文は、このメソッドによって作成され、不要になったオブジェクトすべてを最終化 ("finalize") します。

14. アプリケーションを実行します。

    Bank アプリケーションで、一致する口座すべてをリスト ボックスに表示したダイアログ ボックスが開きます。

15. このダイアログ ボックスをキャンセルして、Bank アプリケーションを閉じます。

16. [アニメート] メニューの [ブレークポイント] をクリックして、[プロジェクト内のすべてをクリア] を選択します。

これで、チュートリアルは終わりです。

7.5 まとめ

このチュートリアルでは、次のことを学習しました。

- おもなコレクション型
- 固有データ
- ディクショナリ
- CallBack
- コレクションの繰返しメソッド

第8章　例外取り扱いのチュートリアル

このチュートリアルでは、Object COBOL のクラス ライブラリにある例外取り扱い機構の使い方について説明します。例外は、オブジェクトがエラーをトラップしたときに発生します。 このチュートリアルは次のセッションからできています。

1. 例外の発生
2. 例外ハンドラの登録
3. 例外ハンドラの作成
4. まとめ

学習時間：15分

8.1 例外の発生

オブジェクトが、エラーをトラップすると例外が発生します。提供されたクラス ライブラリ内のオブジェクトは、およそ 90 の例外条件を定義します。作成したオブジェクトに独自の例外条件を定義することができます。

ExceptionManager クラスは、デフォルトのフォルト例外ハンドラを含む、実際に例外を処理するための論理のほとんどを提供します。デフォルト例外ハンドラは、オブジェクトで例外が発生すると、エラー メッセージを表示してアプリケーションをシャット ダウンします。しかし、アプリケーションは、各自の例外ハンドラを定義して、それらを個々のクラスまたはオブジェクトに接続し、その動作を再定義できます。

オブジェクトは、それ自体に "raiseException" メッセージを送信して、パラメータとしてエラー番号を渡すことによって、例外を発生させます。 Base クラスは、"raiseException" メソッドを実現するため、これはすべてのオブジェクトによって継承されます。

ここでは前の章で紹介した Bank アプリケーションを使って、例外の発生方法の例を示します。Account オブジェクトはエラーをトラップすると、例外を発生することによって応答します。例えば、初期残高が 1000 ドルより少ない高金利預金口座を開設しようとすると、エラー条件が起こります。

例外発生コードのアニメート

1. NetExpress 統合開発環境を開始して、プロジェクト resbank.app をロードします。
2. プロジェクト resbank.app をリビルドします。

   次に、HighRateAccount クラス内の実行するメソッドを検索し、これらのメソッドにブレークポイントを設定します。

3. [ブラウザ] ウィンドウを開きます。

4. [ブラウザ] ウィンドウで HighRateAccount を選択します。

5. テキスト エディタに OpenAccount メソッドをロードします。

6. "openAccount" の最初の文、および最後の文 (**exit method**) にもブレークポイントを設定します。

7. [アニメート] メニューの [アニメート開始] をクリックします。

   実行準備ができた bankapp.int を示すテキスト ウィンドウが表示されます。

8. アプリケーションを実行します。

   制御は Animator からアプリケーションに渡されます。

9. [銀行] ウィンドウが表示されたら、初期残高 0.00 で高金利預金口座を開きます。

10. 制御は Animator に戻ります。

    実行点は、"openAccount" メソッドの最初の文 (**if lsInitialValue < 1000**) にあります。

11. 最初の文をステップ実行します。

    テストが成功します。HighRateAccount クラスが、例外条件を検出しました。

12. 次の文 (**invoke ExceptionManager "queryMessageFile"...**) をステップ実行します。

    Account クラスは、初めて新しい口座を作成すると、メッセージ ファイル account.err を登録します (この処理を実行するコードは、 Account クラスの "setNextAccountNumber" メソッドで確認できます)。ここには、表示対象のデフォルト例外ハンドラのエラー メッセージが含まれます。登録では、account エラーに使用される、ある範囲のエラー番号が割り当てられます。

    "queryMessageFile" メッセージは、数字オフセットを返して、エラー番号範囲の始まりを示します。

13. 次の文 (**add insufficientFundsForAccount to lsErrorNumber**) をステップ実行します。

    実際のエラー番号は、レベル 78 のデータ項目、insufficientFundsForAccount によって定義されます。これは、エラー ファイル内のエラー番号に一致します。実際のアプリケーションは、多数のソースからのクラスからなる場合が多く、各クラスは独自のエラー メッセージ セットを所有しています。

    account エラー メッセージ ファイルのオフセット (前のステップで返された値) をエラー番号に加算すると、固有のエラー番号が作成されます。例外ハンドラは、この番号によって、エラー ファイルとメッセージ番号の両方を識別できます。

14. 次の文 (**invoke self "raiseException"...**) で、例外が発生します。

15. アプリケーションを実行します。

    "raiseException" メッセージを送信すると、 Base にあるメソッド "raiseException" が実行されます。このメソッドは、このオブジェクトに例外ハンドラが登録されているかどうかを確認します。

    この場合、HighRateAccount オブジェクトに例外ハンドラが登録されていて、エラーを示すメッセージ ボックスが表示されます。

16. メッセージ ボックスで [OK] ボタンをクリックして、終了します。

    実行は、"openAccount" の最後にある exit method 文に設定されたブレークポイントで停止します。 例外ハンドラが **stop run** 文によってアプリケーションを閉じないかぎり、実行は常に例外が発生した文に続くポイントから再開されます (これは、例外ハンドラのデフォルトの動作です)。

    この場合、例外ハンドラは、返されたオブジェクト参照をヌルに設定します。新しい account を要求するアプリケーション部分は、参照がヌル値かどうかをチェックして、"openAccount" が正常に実行されたかどうかを確認します。

17. [銀行] ウィンドウの [預金口座] メニューの [終了] をクリックします。

8.2 例外ハンドラの登録

前のセッションでは例外を発生させる方法を示しました。ここでは、オブジェクトについて例外ハンドラを登録する方法について説明します。例外ハンドラは、例外番号と例外を発生するオブジェクトをパラメータとして受け取ります。

例外ハンドラは、クラスまたはインスタンスオブジェクトとして登録できます。例外ハンドラをクラスに対して登録した場合、そのクラス自身に加え、クラスのすべてのインスタンスによって発生した例外について呼び出されます。

例外ハンドラは、2 つのステップで登録します。

1. 例外メソッドのための CallBack を作成する
2. ExceptionHandler を使ってオブジェクトに対して CallBack を登録する

次に Bank アプリケーションにある Account クラスに関する例外ハンドラを設定するためのコードをアニメートします。

例外ハンドラコードのアニメート

1. NetExpress 統合開発環境を開始して、プロジェクト resbank.app をロードします。
2. プロジェクト resbank.app をリビルドします。
3. [ブラウザ] ウィンドウを使用して、MainApplication クラス (bankapp.cbl) の "initialize" メソッドを検索します。
4. タグ B005 (**invoke CallBack "new"...**) の下にある最初の文にブレークポイントを設定します。
5. Animator を開始します。
6. アプリケーションを実行します。

   実行はブレークポイントで停止します。
7. 次の文をステップ実行します。

   self のメソッド (BankApplication インスタンス) である、実行処理メソッド "onExceptionAccountError" に CallBack が作成されます。
8. 次の文 (**invoke ExceptionManager "register"...**) をステップ実行します。

   Account クラスの例外ハンドラとして、accountErrorMethod (BankApplication にある "accountError" の CallBack) が、登録されます。Account クラス、または Account クラスのインスタンスによって発生した例外は、このメソッドに渡されます。
9. 残りの文を **exit method** までステップ実行します。

   これらの文は、他の account クラスによって発生した例外に対して、同じ例外ハンドラを登録します。

これで、チュートリアルのこの部分は終わりです。次のセッションはこのセッションから続けて説明するので、Animator を現在の状態のままにしておいてください。

8.3 例外ハンドラの作成

ここでは、例外ハンドラ メソッドを作成して、オブジェクトで発生した例外を処理する方法を示します。 Bank アプリケーションがアカウント クラスからの例外を処理するために使用する例外ハンドラについて説明します。 高金利預金口座を開くことができない場合、例外ハンドラは、"openAccount" メソッドにヌル オブジェクト ハンドルを返します。

HighRateAccount クラスの "openAccount" メソッド コードは、アカウント オブジェクトへの有効なハンドルの代わりに、例外ハンドラからヌル オブジェクト ハンドルを返します。例外ハンドラは、何か他のものを実行するよう

に指定できます。たとえば、初期調査が不十分であるために高金利預金口座を開けない場合は、代わりに Savings Account を返すことができます。

注記 : この説明は、読者が前のセッションから続けて作業していることを前提としています。

例外ハンドラのアニメート

1. [検索] メニューの [検索] をクリックして、タグ B100 の下の文を検索します。[検索] バーの [検索] フィールドに B100 と入力してから、[検索] バーの [すべて] ボタンをクリックします。処理が終了したら、[表示] メニューの [ツールバー－検索/置換] をクリックして、[検索] バーを削除できます。

2. 下の文 (**set nullReference to null**) にカーソルを移動して、ブレークポイントを設定します。

   これで、ブレークポイントが設定されたため、account 例外ハンドラが起動するたびに実行は停止します。

3. アプリケーションを実行します。

4. 初期残高 0 で James の高金利預金口座を開きます。

   HighRateAccount クラスで例外が発生して、Animator は例外ハンドラの最初の文で停止します。

5. 最初の文をステップ実行します。

   パラメータ nullReference は、最終的に、アプリケーションが作成しようとしている account へのオブジェクト参照として返されます。実行を進めると、このパラメータが返されるパスを確認できます。

6. 次の文 (**invoke ExceptionManager "queryMessageFile...**) をステップ実行します。

   例外ハンドラでは、受信したエラー番号が、処理方法のわかる型かどうかを確認する必要があります。 確認作業は、account エラーのオフセットを取得することによって開始します。

7. 次の文 (**subtract lsOffset from lnkErrorNumber**) をステップ実行します。

   このエラーのオフセットが、ExceptionManager から返されるオフセットに一致する場合、違いは、bank account のエラー番号のどれかに対応する数字になります。

8. 次の文 (**invoke SymbolTable "new"...**) をステップ実行します。

   この文は、account を扱うビジネス ロジックがインタフェースに情報を通知するために使用する機構の一部です。

9. 次の文 (**evaluate lnkErrorNumber**) をステップ実行します 。

   例外ハンドラはここで、適切な処置をとるために発生した例外の型を判別する必要があります。 処理方法

のわからない例外の型である場合、例外ハンドラは再び例外を発生させます (**when other** 文の下のコードを参照)。

10. 次の文 (**when insufficientFundsForAccount**) をステップ実行します。

    これが、発生した例外です。エラー番号はすべて、レベル 78 のデータ項目として、コピーブック accinfo.cpy に定義されています。

11. **exit method** 文までアプリケーションを実行します。

    Bank アプリケーションには、ユーザに対して新しい account の作成に失敗したことを警告するメッセージボックスが表示されます。

12. メッセージ ボックスで [OK] ボタンをクリックして、終了します。

13. **exit method** 文をステップ実行します。

    実行は、HighRateAccount "openAccount" メソッドの **exit method** に戻ります。例外を発生させた文は、戻りパラメータとして lsAccount をもつことがわかります。このパラメータは、今、実行した例外メソッドによって、ヌルに設定されています。

14. **exit method** 文をステップ実行します。

    実行点は、BankApplication の "onSignalOpenAccount" メソッドに戻って、**end-evaluate** 文にあります。このコードは、account が作成されていない可能性を認識し、新しい account のオブジェクト参照がヌルかどうかをテストして確認します。ヌルの場合、実行はこのメソッドの最後にジャンプします。

    例外を正常にトラップした場合、最終的に例外がなかった場合と同じポイントから実行が再開されます。アプリケーションに例外処理をビルドするときは、このことを覚えておいてください。 このため、例外の可能性があるポイントでは、コードは、例外が発生したかどうかをチェックする必要があります。 ヌル参照が戻ったかどうかをチェックするのは、使用できるテクニックの 1 つです。

これで、例外処理に関するチュートリアルは終わりです。Animator はいつでも停止できます。

8.4 まとめ

このチュートリアルでは、次のことを学習しました。

- 例外の発生
- 例外ハンドラの登録
- 例外ハンドラの作成

第9章　GUI プログラミングのチュートリアル

9.1 Phonebook の概要

このチュートリアルでは、GUI アプリケーション テンプレートを使用して、単純な Phonebook アプリケーションを作成します。 このチュートリアルを開始する前に、「入門書」チュートリアルおよびオンラインヘルプの「ブラウザ」チュートリアルを読むことをお勧めします。NetExpress 統合開発環境 (Integrated Development Environment: IDE) を使用してプロジェクトをビルド、実行、および参照する方法についての詳細について説明しています。 『入門書』およびオンラインの「ブラウザ」チュートリアルにリンクするには、[関連トピック] ボタンをクリックしてください。

このチュートリアルは、次のトピックで構成されています。順番に進んでください。一度にすべてのチュートリアルを実行する必要はありません。中断する場合は、作業中のチュートリアルを完了して、phonebook チュートリアル プロジェクトを閉じ、すべての変更が保存されたことを確認して、確実に次のチュートリアルから再開できるように準備してください。

1. phonebook の概要
2. インタフェース出力シグナルの定義
3. [phonebook] メニューの定義
4. [phonebook] ツールバーの定義
5. ダイアログ ボックスの追加
6. クライアント領域の定義
7. ファイルのオープンと保存
8. レコードの追加、削除、および編集
9. ソート機能の追加
10. 入力シグナルへの応答

9.1.1 Phonebook について

Phonebook アプリケーションでは、デスクトップのウィンドウに電話帳が表示されます。この電話帳は、COBOL 行順編成ファイルに保持されています。各レコードは、次のフィールドからなります。

姓
名前

ID
部課名
電話番号

完了したアプリケーションを使用すると、エンドユーザは、次のことを実行できます。

- マスター電話帳を開く (環境変数によって判断され、アプリケーションの前に設定されたファイル名が実行されます)
- ファイル名を指定して、ユーザ電話帳を開く
- 現在開かれている電話帳のレコードすべてを表示する
- 任意のフィールドを検索することによって、レコードの表示を制限する
- レコードを追加、編集、または削除する
- 電話帳に変更を保存する

エンドユーザは、マスター電話帳への変更を保存できないため、この電話帳が開かれている場合、エンドユーザは、別のファイルに変更を保存する必要があります。

Phonebook の実行例

1. NetExpress 統合開発環境を開始して、phonebook アプリケーション(phone¥phone.app)をロードします。
2. [プロジェクト] メニューの [リビルド] をクリックして、プロジェクト内のファイルすべてをコンパイルします。
3. [アニメート] メニューの [実行] をクリックするか、またはツールバーの [実行] アイコンをクリックして、プロジェクトを実行します。
4. [ファイル]、[開く]、[電話帳ユーザ] をクリックします。

   [電話帳マスター] オプションは、設定された環境変数からそのファイル名が取得されるため、使用できません。 このオプションの動作を確認するには、プロジェクトに含まれる readme.txt ファイルを検討してください。
5. [開く] ダイアログから tel.lst を選択して、[OK] をクリックします。

   phonebook のレコードが、[電話帳] ウィンドウに表示されます。レコードの編集、追加、および削除機能 ([レコード] メニューまたはツールバーから使用可能) と検索機能 ([ファイル] メニューとツールバーから使用可能) を試行して、アプリケーションによって提供されている機能を確認してください。

9.1.2 このチュートリアルで使用されるクラス

phonebook アプリケーションは、8 つの Object COBOL クラスからなります。すべての単一クラスを詳しく説明する代わりに、次の 3 つに焦点を当てて説明します。

- MainWindow クラス (tpltwin.cbl)
- MainApplication クラス (tpltapp.cbl)
- RecordDetailsDialog クラス (recrdbox.cbl)

このチュートリアルでは、GUI アプリケーション テンプレートに提供されたデフォルト バージョンに追加することによって、MainWindow クラスを作成します。 他のクラスは完全にコード化された状態で提供されているため、コードのいくつかを検討する場合でも、それを作成することはありません。 MainWindow クラスに追加する必要があるコードはすべて、このチュートリアルの一部として含まれているため、それをそのままコピーして、アプリケーションに貼り付けることができます。

Phonebook で使用されるその他のクラスは、次のとおりです。

- AboutBox クラス (tpltabt.cbl)
- PBSearchDlg クラス (pbsrch.cbl)
- TelRecord クラス (telrec.cbl)
- TelBook クラス (telbook.cbl)
- Query クラス (query.cbl)

Phonebook には、1 つの手続き COBOL プログラムがあります。

- トリガ (tplttrig.cbl)

トリガは、テンプレートの一部であり、phone アプリケーションを開始するために実行するプログラムです。phonebook トリガは、GUI テンプレートのトリガとまったく同じであり、最初からアプリケーションをビルドした場合でも、何も変更する必要がありません。

9.1.3 チュートリアルの開始

編集用に独立したバージョンの phonetut¥Phonetut.app プロジェクトが用意されています。チュートリアル バージョンと完全作業バージョンで異なるファイルは、次のとおりです。

tpltwin.cbl

tpltres.dlg

tpltmenu.mnu

mfres.h

mfres.cpy

9.2 新しいインタフェース出力シグナルの定義

9.2.1 シグナルの定義

GUI アプリケーション テンプレートは、コンポーネントとシグナルを使用して、アプリケーションのビジネス ロジックをそのユーザインタフェースから分離します。 テンプレートを使用して作成されたアプリケーションは、2 つの主要コンポーネントによってビルドされています。各コンポーネントは、次の Object COBOL クラスによって表わされます。

- MainWindow クラスは、インタフェース コンポーネントです。
- MainApplication クラスは、ビジネス ロジック コンポーネントです。

この節では、MainWindow コンポーネントが送信できるすべてのシグナルを定義します。1 つのシグナルは最大 30 文字の長さの、任意のヌル終了定数にできます。 シグナルのコード化を簡単にするために、シグナルはすべてコピーファイル内のレベル 78 データ項目に割り当てられており、シグナルを使用するコードはすべて、レベル 78 のデータ名によってそのシグナルを参照します。 データ名の入力を間違えるとプログラムの作成時に検出されますが、シグナル名を示す定数を間違えた場合は、プログラムを実行するまでその間違いは検出されません。

MainWindow コンポーネントによって送信されたシグナルはすべて tpltwin.cpy で割り当てられて、MainApplication コンポーネントによって送信されたシグナルは tpltapp.cpy で割り当てられます。チュートリアル ディレクトリ内のコピーファイルでは、レベル 78 の項目すべてがすでに自動的に割り当てられています。 割り当てられていない場合は、完全コード化された MainApplication (tpltapp.cbl) をエラーなしでコンパイルすることはできません。

シグナルは一方向であり、シグナルによってデータを返すことはできません。インタフェースとビジネス ロジックの間の通信は、インタフェース要求シグナルにアプリケーション実行シグナルが続く形式をとります。 たとえば、インタフェースが検索シグナルを送信する場合、その結果は、レコード表示シグナルを送信するビジネス ロジックによって返されます。

出力シグナル定義コード

1. Phone アプリケーション MainWindow クラス (tpltwin.cbl) の "definesignals" メソッドを検索します。
2. 下のコードをコピーして、"defineSignals" メソッドの "ユーザコード" と "ユーザ終り" コメントの間に貼り付けます。

```
define signal openmaster-signal

define signal saveas-signal

define signal search-signal

define signal sort-signal

define signal import-signal

define signal edit-signal

define signal add-signal

define signal delete-signal
```

9.2.2 Define Signal 句

各シグナルは、動詞句 DEFINE SIGNAL によって定義されます。これは、新しい動詞句であり、NetExpress のボキャブラリ機能を使用して作成されています。GUI テンプレートには、コードを簡素化し、さらに読みやすくするための新しい動詞句の拡張ボキャブラ リがあります。

```
define signal openmaster-signal
```

上記のコードは、次のコードと等価です。

```
invoke self "defineOutSignal" using openmaster-signal
```

上記で追加した新しいコードに定義された各シグナルは、tpltwin.cpy のレベル 78 のデータ項目に割り当てられています。このコピーファイルは、これらのシグナルへの応答を定義するすべてのプログラムによって使用されます。

各 DEFINE SIGNAL は、指定されたシグナルをこのコンポーネントで使用できるようにします。使用する予定のシグナルを定義しないと、シグナルを送信しようとしたときに実行時エラーが発生します。

9.3 [Phonebook] メニューの定義

このメニューは、リソース ファイル tpltmenu.mnu に定義されます。GUI アプリケーション テンプレートに基づくアプリケーションのリソースはすべて、tpltres.rc に含まれています。

9.3.1 メニューのマッピング

メニュー バーの全体イメージを決定する Windows リソースを更新しました。これらの更新内容を有効にするには、書くメニュー項目をメソッドにマッピングします。このチュートリアルの後半では、新しいメソッドを MainWindow クラスに追加しますが、ここでは、特定のメニュー項目がクリックされたときにどのメソッドが起動するかを指示するだけです。

メニュー項目は、MainWindow クラスの "defineMenuMap" メソッドによってマッピングされます。

入力シグナル定義コード

1. [ブラウザ] ウィンドウを使用して、MainWindow クラスの "defineMenuMap" メソッドに進みます。

   "defineMenuMap" メソッドのデフォルト コードは、[ファイル] メニューの [新規作成] から削除したメニュー項目を定義します。これは、マッピング コードから削除する必要があります。そうしないと、この項目をメソッドにマッピングしようとしたときに、実行時エラーが発生します。

2. 文を検索して削除します。

   ```
   map menuid ID-FILE-NEW at osResource to z"onEventNew"
   ```

3. 下のコードをコピーして、Phone アプリケーション MainWindow クラス (tpltwin.cbl) の "defineMenuMap" メソッドの "ユーザコード" と "ユーザ終り" コメントの間に貼り付けます。

   ```
   map menuId ID-FILE-MASTER at osResource to
                                   z"onEventOpenMaster"
   map menuId ID-FILE-IMPORT at osResource to z"onEventImport"
   map menuId ID-FILE-SEARCH at osResource to z"onEventSearch"
   map menuId ID-FILE-SAVEAS at osResource to z"onEventSaveAs"
   map menuId ID-FILE-SORT at osResource to z"onEventSort"
   map menuId ID-RECORD-ADD at osResource to z"onEventAdd"
   map menuId ID-RECORD-EDIT at osResource to z"onEventEdit"
   map menuId ID-RECORD-DELETE at osResource to z"onEventDelete"
   ```

9.3.2 メニュー マップの機能

"defineMenuMap メソッド" のコードは、動詞句 MAP MENUID に定義されたボキャブラリを使用します。各 MAP MENUID 文は、メニュー項目をメソッドに接続します。これは、メニューのヒントとツール情報が定義されている場合、これらも設定します。これらは、Windows リソースとして、stringtable にも定義されています。

メニュー マッピング コードの追加例

1. [プロジェクト] ウィンドウの tpltres.rc をダブルクリックして、リソース マネージャを起動します。

2. [リソース マネージャ] ウィンドウの [メニュー] エントリをダブルクリックします。

3. ファイル名 tpltmenu.mnu をダブルクリックします。

4. テキスト編集ウィンドウにメニュー ファイルが開きます。このファイルは COBOL コードではなく、Windows のリソース定義コードです。メニュー コードの始まりは、次の行によって示されます。

   ```
   ID_MENUBAR MENU
   ```

   ID_MENUBAR はシンボルであり、プログラムからメニュー リソースを識別できるようにします。MENU は、次の一連の行がメニュー リソースを定義することを示すキーワードです。メニュー定義は、BEGIN および END の各キーワードで囲まれます。 各サブメニューも、メニュー BEGIN および END 内部に入れ子する BEGIN と END によって囲まれます。メニュー項目は、次の形式の行です。

   ```
   MENUITEM "text"                  symbol
   ```

   引用符に囲まれたテキストはウィンドウ上のメニュー項目として表示され、シンボルはメニュー項目を識別するため、プログラムからこの項目にアクセスできます。

5. 行 ID_MENUBAR MENU のすぐ下にある BEGIN から、それに対応する END 文までの行すべてを削除します。

6. 下のコードをコピーして、行 ID_MENUBAR のすぐ下に貼り付けます。

```
BEGIN
  POPUP "ファイル(&F)"
  BEGIN
     POPUP "開く(&O)"
     BEGIN
     MENUITEM "電話帳マスター(&M)",         ID_FILE_MASTER
     MENUITEM "電話帳ユーザ(&U)...",        ID_FILE_OPEN
  END
     MENUITEM "インポート(&I)"              ID_FILE_IMPORT
     MENUITEM "検索(&e)...",                ID_FILE_SEARCH
     MENUITEM "上書き保存(&S)",             ID_FILE_SAVE
     MENUITEM "名付けて保存(&a)...",        ID_FILE_SAVEAS
```

```
        MENUITEM SEPARATOR
        MENUITEM "ソート(&o)"                ID_FILE_SORT
        MENUITEM SEPARATOR
        MENUITEM "終了(&x)",                 ID_APP_EXIT
    END
    POPUP "レコード(&R)"
    BEGIN
        MENUITEM "レコードの追加...",        ID_RECORD_ADD
        MENUITEM "レコードの編集...",        ID_RECORD_EDIT
        MENUITEM "レコードの削除...",        ID_RECORD_DELETE
    END
    POPUP "ヘルプ(&H)"
    BEGIN
        MENUITEM "ヘルプ(&H)",               ID_HELP_HELP
        MENUITEM "製品について(&A)",         ID_HELP_ABOUT
    END
END
```

7. tpltmenu.mnu を保存して、このファイルのテキスト ウィンドウを閉じます。

8. [リソース マネージャ] ウィンドウ (タイトルバー tpltres.rc を持つ) をクリックするか、[ウィンドウ] メニューの [tpltres.rc] をクリックします。

   Windows リソースで使用される各シンボルには、数値を指定する必要があります。これは実際には、各リソースを参照する数値です。 追加した各メニュー項目には、値を指定する必要がある新しい識別シンボルが含まれていました。このシンボル値は、COBOL コピーファイル mfres.cpy にも変換される mfres.h に定義されています。

9. [リソース] メニューの [シンボル] をクリックします ([リソース マネージャ] ウィンドウがアクティブな場合は、新しいメニューが IDE メニューバーリソース] に表示されます)。

10. これにより、[リソースシンボル] ダイアログ ボックスが開きます。ここには、シンボルと数値のリストが表示されます。

11. [新規] ボタンをクリックします。

12. [シンボル] フィールドに、[ファイル] メニューの [Open Master] を示すシンボルである ID_FILE_MASTER を入力し、これに 504 の値を指定します。各シンボルが固有の値を持つ必要がある以外、シンボル値に意味はありません。Window のリソース コンパイラは大文字小文字を区別するため、ID_FILE_MASTER と大文字で入力してください。

13. [適用] ボタンをクリックします。

    このアプリケーションを最初からビルドする場合は、メニュー定義に使用される新しいシンボルすべてをここで追加します。 時間を節約するために、すでに別の新しいシンボルを追加してあるため、[OK] ボタンを押して、ダイアログ ボックスを終了してください。

14. [リソースマネージャ] ウィンドウを閉じて、tpltres.rc への変更を保存するかどうかを尋ねられたら、[はい] とクリックします。

9.3.3 Stringtable 形式

各文字列は、次の形式をとります。

symbol "*Menu hint*¥n*Tool tip*"

文字列のシンボルは、この文字列がヒントであるメニュー項目のシンボルと一致しなければなりません。 たとえば、[レコード] メニューのメニュー項目 [レコードの追加] は、シンボル ID_RECORD_ADD を持ちます。これに一致する文字列 テーブル エントリは、次のとおりです。

```
ID_RECORD_ADD           "新しいレコードの追加¥nレコードの追加"
```

[レコードの追加] メニュー項目が [レコード] メニューで強調表示されると、必ずプロンプト "新しいレコードの追加 " がステータスバーに表示されます。ツールバーを設定する場合 (このチュートリアルの後半で)、マウス カーソルを [レコードの追加] ボタン上にしばらく置くと、ステータスバーに表示されるメニューのヒントに加えて、ボタンの隣に 小さなラベル プロンプト "レコードの追加" が表示されます。

Stringtable 定義の追加例

1. Phone アプリケーションのメニュー リソース (tpltmenu.mnu) を以前のように開きます。

2. 行 STRINGTABLE DISCARDABLE を検索します。

3. BEGIN 文と END 文の間のコードを削除してから、下のコードをコピーし、BEGIN 文と END 文の間に貼り付けます。

```
ID_FILE_MASTER          "マスタ電話帳を開く¥nマスタを開く"
ID_FILE_OPEN            "ユーザ電話帳を開く¥nユーザを開く"
ID_FILE_SEARCH          "電話帳の検索¥n検索"
ID_FILE_SAVE            "現在の電話帳を保存¥n保存"
ID_FILE_SAVEAS          "違う場所または名付けて保存¥n名付けて保存"
ID_APP_EXIT             "電話帳アプリケーションの終了¥n終了"
ID_RECORD_ADD           "新しいレコードの追加¥nレコードの追加"
ID_RECORD_EDIT          "レコード編集¥nレコード編集"
ID_RECORD_DELETE        "レコード削除¥nレコード削除"
ID_HELP_HELP            "ヘルプトピックの表示¥nヘルプ"
ID_HELP_ABOUT           "プロダクト情報¥nプロダクト情報"
```

9.4 [Phonebook] ツールバーの定義

ほとんどの Windows アプリケーションにはツールバーがあります。エンドユーザは、これをショートカットとして使用して、使用頻度の高いメニュー項目を利用できます。 一度メニューを設定すれば、GUI アプリケーション テンプレートによるツールバーの設定は非常に簡単です。ビットマップ エディタ (NetExpress は、デフォルトで Micro Focus Image Editorを使用する) を使用してプッシュボタンを 描画し、アプリケーションのリソースにツールバーをコンパイルし、さらにツールバー上のボタンをメニュー項目にマッピングします。

9.4.1 ツールバーの更新

ツールバーの更新例

1. [リソース マネージャ] ウィンドウを開きます。
2. [ビットマップ] エントリをダブルクリックします。
3. ID_TPLTTOOL をダブルクリックします。

   アプリケーション テンプレートのデフォルト ツールバーを開きます。phonebook アプリケーションのツールバー ビットマップをコピーする代わりに、適切なビットマップが用意されています。

4. イメージ エディタを閉じます。

5. ID_TPLTTOOL を右ボタンでクリックして、コンテキスト メニューの [プロパティ] をクリックします。

6. リソース ファイル名の内容を pbtool.bmp に変更します。

7. [ビットマップのプロパティ] ダイアログ ボックスの [OK] をクリックします。

8. ID_TPLTTOOL をダブルクリックして、イメージ エディタを再び開きます。

   マウス カーソルをウィンドウの一番上にあるビットマップに移動すると、位置標識が変わることを確認できます。ビットマップ内の最初の位置は左上端であり、0, 0 という数字で示されます。ツールバー上にボタンを作成すると、ビットマップは、16 ピクセル幅のより小さいビットマップに分割されます。これらのより小さい各ビットマップは、1 つのボタンを表わします。最初のビットマップは、0, 0 から 15, 14 に続きます。 2 番めのビットマップは、x 位置 16 から始まり、3 番めは 32 という具合に続きます。

9. [リソース マネージャ] ウィンドウを閉じます。

   これで、アプリケーションでしよう可能なツールバー ビットマップが作成されましたが、各プッシュボタンを明示的にツールバーに追加して、メニュー項目にマッピングする必要があります。

10. [ブラウザ] ウィンドウを使用して、MainWindow クラスの "defineToolbarMap" メソッドに進みます。

    このメソッドには、ツールバー ビットマップ内にある最初の 3 つのサブビットマップをツールバーに追加する、デフォルト コードが含まれます。 このコードを、ツールバー上に 6 つのボタンを作成し、それらを適切なメニュー項目にマッピングする新しいコードによって置換します。

11. "defineToolbarMap" メソッドの "ユーザコード" および "ユーザ終り" の各コメント間にあるコードを削除します。

12. 下のコードをコピーして、Phone アプリケーション MainWindow クラス (tpltwin.cbl) の "defineToolbarMap" メソッドの "ユーザコード" と "ユーザ終り" コメントの間に貼り付けます。

```
           map button 3 at osToolbar to menuId ID-FILE-SAVE
                                                  at osResource

       *----セパレータの追加

           invoke osToolbar "addSeparator" returning lsTemp

       *---- [Search File] ボタンをツールバーに追加

           map button 1 at osToolbar to menuId ID-FILE-SEARCH
                                                  at osResource
```

```
*----セパレータの追加
    invoke osToolbar "addSeparator" returning lsTemp
*----ボタンをツールバーに追加
    map button 4 at osToolbar to menuId ID-RECORD-ADD
                                        at osResource
*----ボタンをツールバーに追加
    map button 5 at osToolbar to menuId ID-RECORD-EDIT
                                        at osResource
*----ボタンをツールバーに追加
    map button 6 at osToolbar to menuId ID-RECORD-DELETE
                                        at osResource
*----セパレータの追加
    invoke osToolbar "addSeparator" returning lsTemp
*----ボタンをツールバーに追加
    map button 8 at osToolbar to menuId ID-HELP-HELP
                                        at osResource
```

13. 次の宣言をこのメソッドの一番上にある local-storage 節に追加します。

```
01 lsTemp             object reference.
```

9.4.2 ツールバー マップの機能

各文は、次の形式をとります。

`map button` *`n`* `at osToolbar to menuId` *`symbol`*

MAP BUTTON は、別のボキャブラリ定義動詞句です。これは、ビットマップ上の n 番目 のサブビットマップを、ツールバー上の次のボタンとして追加し、それを symbol によって表わされるメニュー項目にマッピングします。

MAP BUTTON 文の順序は、ボタンがツールバーに表示される順序を決定します。索引 n は、ツールバー ボタンのイメージに使用されるサブビットマップを指定します。

ツールバーとビットマップを追加した結果は、アプリケーションをリビルドして実行することによって確認できます。 新しいメニュー項目またはツールバー ボタンはクリックしないでください。これらのほとんどは、まだ MainWindow に追加していないメソッドにマッピングされています。もしこれらをクリックすると、"doesNotUnderstand" 例外が発生します。ただし、ツールバー ボタンの上のマウス カーソルをゆっくり移動すると、ツール情報とメニューのヒントを表示できます。 これらの情報を読み終えたら、Phonebook の [ファイル] メニューの [終了]をクリックしてください。

9.5 ダイアログ ボックスの追加

9.5.1 phonebook のダイアログ ボックス

この節では、アプリケーションにダイアログ ボックスを追加します。phonebook アプリケーションは、3 つのダイアログ ボックスをまとめて使用します。

- [About(プロダクト情報)] ダイアログ ボックス

これは、GUI アプリケーション テンプレートの一部として提供されています。このダイアログ ボックスは、全体イメージを編集するだけですみます。このボックスをコントロールするコードはすべてテンプレートに組み込まれています。

- [レコードの詳細] ダイアログ ボックス

電話レコードを表示するか、または追加するためのダイアログ ボックス。

- [検索] ダイアログ ボックス

電話帳を検索するためのダイアログ ボックス。

各ダイアログ ボックスでは、次のことを実行する必要があります。

- ダイアログ ボックスに色を付ける。
- 提供されているテンプレート ダイアログ ボックス クラスから、ダイアログ ボックスのクラスを作成する。
- `MainWindow` に、ダイアログ ボックスをコントロールするためのコードを追加する。

ダイアログ ボックスを追加する作業はどれも似ているため、このチュートリアルでは、[検索] についてのみ詳しく説明しま す。

まず、リソース マネージャから Dialog Editor を起動し、これを使用してダイアログ ボックスに色を付けます。

9.5.2 ダイアログ ボックス クラスの作成

Windows リソースに色を付けて、ダイアログ ボックスの全体イメージを定義したら、Object COBOL クラスを作成

してその動作を定義できます。このクラスはすでに自動的に作成されていて、PBSearchDlg (ファイル名 pbsrch.cbl) という名前が付けられています。これを表示するには、[Project] ウィンドウで pbsrch.cbl をダブルクリックします。GUI アプリケーション テンプレートを使用する場合は、最初からこれらのクラスを作成する必要はありません。ファイル tpltdbox.cbl を新しい名前でコピーし、独自のダイアログ ボックスを作成するための開始点としてこのファイルを使用できます。

PBSearchDlg の "create" メソッドまでページダウンします。ダイアログ ボックス上の各コントロールへのオブジェクト ハンドルを取得するために、これに似たコードが使用されます。

```
move IDOK to lsResourceId

invoke self "getObjectFromId" using lsResourceId

                        returning lsPushbutton
```

"create" メソッドの最初の部分は、[OK] ボタンのオブジェクト ハンドルをフェッチして、クリックされたイベントをダイアログ ボックス上の論理イベント、dialog-ok にマッピングします。エンドユーザが [OK] ボタンをクリックすると、ダイアログ ボックスで dialog-OK イベントが発生します。このチュートリアルの後半でダイアログ ボックスを作成し、表示する MainWindow コードを調べると、MainWindow が dialog-ok イベントをメソッドにマッピングしていることがわかります。

クリックされたイベントを処理する方法としては、回りくどいように思われます。しかし、ダイアログ ボックス コードを作成するプログラマと MainWindow コードを作成するプログラマが同じであっても、作業を簡素化できます。ダイアログ ボックス コードを作成する場合は、特定イベントが発生したときに別のオブジェクトでどのメソッドを起動するかを前もって考える必要はありません。

MainWindow でコードを作成する場合は、ダイアログ ボックスのコントロールに関する特別な知識は必要ありません。エンドユーザが [OK] をクリックすると、ダイアログ ボックスで dialog-OK イベントが発生するだけです。たとえば、ダイアログ ボックス内にリストボックスがある場合は、リスト内のエントリのダブルクリックも dialog-ok にマッピングしてください。この処置は、他のどこかにあるコードを変更しなくても実行できます。

同様に、[キャンセル] は、dialog-cancel イベントにマッピングされます。新しい論理イベント (たとえば、dialog-apply) を定義することによって、他のダイアログ ボックス イベントにもこの基本概念を拡張できます。規則により、ダイアログ イベントは、tpltdial.cpy のレベル 78 データ名に割り当てられています。レベル 78 の各データ名には、一意の数字 ID が与えられています。

ダイアログ ボックスの色付け例

1. [プロジェクト] ウィンドウの tpltres.rc をダブルクリックして、リソース マネージャを起動します。
2. [リソース] ウィンドウの [ダイアログ] をダブルクリックします。
3. [ダイアログ グループ] をダブルクリックして、ダイアログ エディタを起動します。

4. ダイアログ エディタが起動して、選択対象のダイアログ ボックスのリストが表示されます。

   これらのダイアログ ボックスは、リソースで現在定義されているものです。DLG_ABOUT は [バージョン情報] ダイアログ ボックス、IDD_RECORD_DETAILS は [レコードの詳細] ダイアログ ボックスです。

5. IDD_RECORD_DETAILS を選択します。

6. [ダイアログ エディタ] ウィンドウの一番上にある [シンボル] フィールドから、このダイアログ ボックスのシンボルが IDD_RECORD_DETAILS であることがわかります。他のコントロールをマウスでクリックすると、それらに割り当てられたシンボルを表示できます。各エントリ フィールドとプッシュボタンにはシンボルが割り当てられているため、このダイアログ ボックスのコードは、Windows リソースからこれらにアクセスできます。

7. [編集] メニューの [新規ダイアログ] をクリックします。

   ダイアログ エディタは、新しいダイアログ ボックスを作成します。

8. [キャプション] フィールドに "検索" と入力し、[ダイアログ シンボル] フィールドに "IDD_SEARCH" と入力して、Enter キーを押します。

9. これを作成するには、ダイアログ エディタのフローティング ツールバーから各自のダイアログ ボックスに、コントロールをドラッグします。 フローティング ツールバー上のボタンの機能がわからない場合は、マウス カーソルをそのボタン上に移動して、ツールのヒントを表示します。

   コントロールをマウスでクリックし、表示される選択ボックス上の小さな四角形をドラッグすると、コン

トロールのサイズを変更できます。

各コントロールには、下の表で示されたシンボルとキャプションを指定してください。あるいは、上記のダイアログ ボックスのピクチャをクリックして、各コントロールとそのシンボルを確認することができます。

コントロール	キャプション	シンボル
編集フィールド	*なし*	IDC_SEARCH_ENTRY
オプション ボタン 1	名前(&F)	IDC_RAD1_FORENAME
オプション ボタン 2	姓(&u)	IDC_RAD1_SURNAME
オプション ボタン 3	ID(&I)	IDC_RAD1_ID
オプション ボタン 4	部署(&D)	IDC_RAD1_DEPT
オプション ボタン 5	内線(&P)	IDC_RAD1_PHONE
プッシュ ボタン 1	OK(&O)	IDOK
プッシュ ボタン 2	キャンセル(&C)	IDCANCEL
テキスト ラベル	検索(&S)	*なし*

[検索] フィールドのテキスト ラベルを追加したら、[オプション] メニューに進んで、[前置語コントロール] をオフに切り替えます。アプリケーションからテキスト ラベルにアクセスすることはないため、テキスト ラベルにシンボルを指定する必要はありません。

10. 処理が終了したら、[ファイル] メニューの [終了] をクリックし、変更を保存するかどうかを尋ねるダイアログ ボックスすべてに対して [OK] とクリックします。最後に [リソース] ウィンドウを閉じてください。

9.5.3 ダイアログ ボックスの表示

次に、エンドユーザが [ファイル] メニューの [検索] をクリックしたときに [検索] ダイアログ ボックスを表示するコードを追加します。このメニュー項目は、"onEventSearch" と呼ばれるメソッドにマッピングされています。このことは、MainWindow メソッド "defineMenuyMap" のコードを見て、そこに次の文が含まれることをチェックすると確認できます。

```
map menuId ID-FILE-SEARCH at osResource to z"onEventSearch"
```

ID-FILE-SEARCH は、メニュー リソース ファイル tpltmenu.mnu の [ファイル] メニューにある [検索] を識別するシンボルです。

9.5.4 "onEventSearch" の説明

"onEventSearch" メソッド コードのコメントは、このコードが実行する 3 つのタスクを示しています。

- ダイアログ ボックスのインスタンス化

"fromResource" メッセージは、PBSearchDlg クラスの新しいインスタンスを作成します。これは、新しいダイアログ ボックスになります。

- MAP EVENT 文句に定義されたボキャブラリを使用する、メソッドへのダイアログ ボックス論理イベントのマッピング

dialog-ok イベントは "onDialogSearchOk" にマッピングされ、dialog-cancel イベントは "onDialogCancel" にマッピングされます。

- ダイアログ ボックスの表示

"onDialogCancel" メソッドは、MainWindow クラスにすでに含まれており、ダイアログ ボックスを自動的に廃棄します。ほとんどのダイアログ ボックスでは、このメソッドで十分です。 しかし、アプリケーションによっては、MainWindow が特定のダイアログ ボックスを表示するときに、いくつかの状態情報を記録する必要があります。 この場合は、これらのダイアログ ボックスに対して、ダイアログ ボックスがキャンセルされたときに状態情報をリセットする特殊なキャンセル メソッドを記述するか、または "onDialogCancel" メソッドを拡張する必要があります。

"onDialogSearchOK" メソッドを MainWindow に追加して、エンドユーザがダイアログ ボックスの [OK] ボタンをクリックしたときにイベントを処理できるようにする必要があります。

"onEventSearch" メソッドの追加例

1. 下のコードをコピーし、Phone アプリケーションの MainWindow クラス (tpltwin.cbl)の、END-OBJECT コード (プログラムの終わり近く) の直前に貼り付けます。

```
*>------------------------------------------------------
*> ウィンドウ イベント メソッド
*> 選択中のファイルで起動し、検索する
*>------------------------------------------------------
 method-id. "onEventSearch".
 local-storage section.
 01 lsId                  pic x(4) comp-5.
```

```
01 lsDialog              object reference.

procedure division.

*----Windows リソースからダイアログ ボックス オブジェクトの例をあげる

    move IDD-SEARCH to lsId

    invoke PBSearchDlg "fromResource" using self osResource lsId

                          returning lsDialog

*----ダイアログを OK にマップし、メソッドへのイベントをキャンセルする

    map event dialog-ok upon lsDialog

                to method z"onDialogSearchOk" upon self

    map event dialog-cancel upon lsDialog

              to method z"onDialogCancel" upon self

*----ダイアログを表示する

    invoke lsDialog "create"

    invoke lsDialog "show"

    exit method.

end method "onEventSearch".
```

このメソッドで使用される PBSearchDlg クラスは、まだ Class-Control 段落で宣言されていません。

2. MainWindow クラスの一番上に進んで、Class-Control 段落を検索します。この段落の一番下にある "ユーザコード" と "ユーザ終り" コメントの間に、次の宣言を追加します。

```
PBSearchDlg is class "pbsrch"
```

ファイル名 （宣言の引用符に囲まれた部分）は小文字で入力する必要があります。

9.5.5 "onDialogSearchOk" の説明

"onDialogSearchOk" メソッドは、次のことを実行します。

- 検索を開始したダイアログ ボックスのオブジェクト ハンドルを抽出します。メソッドがイベントによって起動されるたびに、イベント オブジェクトがパラメータとして渡されます。イベント オブジェクトに

問い合わせると、イベントを発生させたオブジェクトを含む、イベントに関する詳細を取得できます。

- ダイアログ ボックスへのデータ入力を抽出します。エンドユーザが行なった選択はすべて、ダイアログ ボックスのコントロール内に保持されています。PBSearchDlg クラスには、MainWindow がこのデータにアクセスできるようにするメソッドが含まれます。

- SymbolTable にデータを挿入します。SymbolTable を使用すると、シグナルとともに任意の数のパラメータを渡すことができます。シグナルの受信側は、パラメータが保存されているシンボルを指定することによって、必要な各パラメータをフェッチします。シグナルの送信側も受信側も、送信されたパラメータの数や順序を考慮する必要はありません。

- シグナルを送信します。

- SymbolTable とその内容を最終化します。 オブジェクトは、メモリ リークを防止するため、不要になったら必ず廃棄する必要があります。

[検索] ダイアログ ボックスはモードレスであるため、メソッドの最後には開かれたままになります。ダイアログ ボックスがモード付きの場合は、メソッドの最後にメッセージ "hide" および "finalize" を送信し、デスクトップからそのダイアログ ボックスを削除して廃棄します。

ここでアプリケーションをリビルドして実行する場合は、これらのメソッドの動作を確認できます。Phonebook [ファイル] メニューの [検索] をクリックして、新しいダイアログ ボックスを表示してください。情報を入力して、[OK] ボタンをクリックします。検索結果を処理するためのコードをまだ挿入していないため、何も起こりません。ダイアログ ボックスを取り消すには、[キャンセル] ボタンを クリックします。アプリケーションを閉じるには、Phonebook の [ファイル] メニューの [終了]をクリックします。

"onDialogSearchOk" メソッドの追加例

下のコードをコピーし、Phone アプリケーションの MainWindow クラス (tpltwin.cbl)の、END-OBJECT コード (プログラムの終わり近く) の直前に貼り付けます。

```
*>------------------------------------------------------
*> ウィンドウ イベント メソッド
*> 選択中の OK で起動し、ダイアログ ボックスを検索する
*>------------------------------------------------------
 method-id. "onDialogSearchOk".
 local-storage section.
 01 lsDialog              object reference.
```

```
 01 lsDataBlock            object reference.
 01 lsSearchText           object reference.
 01 lsSearchType           object reference.
 01 lsType                pic x.
 01 i                     pic x(4) comp-x.

 linkage section.
 01 lnkEvent              object reference.
 procedure division using lnkEvent.
*----イベントが発生したダイアログ ボックスを処理する
    invoke lnkEvent "getObject" returning lsDialog
*----データをダイアログ ボックスに入力する
    invoke lsDialog "getSearchText" returning lsSearchText
    invoke lsDialog "getSearchType" returning lsSearchType
*----シンボル表にデータを挿入する
    invoke SymbolTable "new" returning lsDataBlock
    put value lsSearchType into table lsDataBlock
                        at symbol searchtype-key
    put value lsSearchText into table lsDataBlock
                        at symbol searchstring-key
    invoke wsEventManager "setSystemBusy"
*----検索シグナルを送る
    signal search-signal lsDataBlock
*----シンボル表と内容を完成させる
    invoke lsDataBlock "deepFinalize" returning lsDataBlock
```

```
        exit method.

    end method "onDialogSearchOk".
```

これで、ダイアログ　ボックスの追加方法とそのコントロール方法についての説明は終わりです

9.6 クライアント領域の定義

9.6.1 phonebook のクライアント領域

ウィンドウのクライアント領域は、ほとんどのアプリケーションの主要情報領域です。クライアント領域に情報を挿入する方法は、いくつかあります。

- リスト ボックスまたは複数行エントリ (MLE) フィールド
- ツリー ビュー
- サブペイン

サブペインは自由に使用できます。サブペインの独自のサブクラスを作成して、"paint" メソッドを上書きできます。クライアント領域にはテキストまたはグラフィック、あるいはこの両方を表示して、位置指定、サイズ、色、およびフォントを完全にコントロールできます。 ただし、Phonebook などの単純なデータ処理アプリケーションの場合は、サブペインをコード化してもあまり意味がありません。

Phonebook のクライアント領域には、レコードのリストが表示されます。各レコードは 1 つの行に収まります。 このようなアプリケーションでは、はるかに少ないコストで、リストボックスに必要な機能すべてが示されます。

アプリケーションのクライアント領域は、MainWindow の "defineClientPane" メソッドで設定されます。MainWindow の "defineClientPane" メソッドを検索してください。現在、これには次の 1 つの文が含まれています。

```
invoke Subpane "new" using lnkParent returning lnkSpace
```

このコードを、リスト ボックスとしてクライアント領域を定義するコードで置換します。

9.6.2 "defineClientPane" の説明

このコードは、次のことを実行します。

単一の選択リストボックスを作成する。 単一の選択と複数選択の、2 つの型のリスト ボックスがあります。単一選択リスト ボックスを使用すると、エンドユーザは、一度に 1 つのリストボックス エントリを選択できます。このリスト ボックスは、クラス SingleSelectionListBox によって表示されます。 リスト ボックスの全体イメージを設定して、黄色いバックグラウンドを指定し、ウィンドウのクライアント領域に必ず正確に収まるようにする。 リスト ボックスのタブ位置を設定する。

5 つのタブ位置を設定するために、コードは 5 つの要素を持つ配列を作成し、値を設定してから、"setTabStops" メソッドのパラメータとしてこの配列を使用します。タブ位置設定は、ピクセル数で指定します。 配列などのコレクション オブジェクトの詳細については、コレクション、固有機能、およびディクショナリに関するチュートリアルを参照してください。

リスト ボックス イベントをメソッドにマッピングする。

エンドユーザがリスト ボックス内のレコードを選択すると、"onEventRecordSelect" メソッドが起動されます。これにより、Phonebook アプリケーションは、機能の編集や削除を実行できますが、この処置は通常、まずレコードを選択してからでないと、その削除や編集を実行できないため使用不能です。 エンドユーザがリスト ボックスのレコードをダブルクリックすると、"onEventEdit" メソッドが起動されて、[Edit record] ダイアログ ボックスが表示されます。 このメソッドには、Array および SingleSelectionListBox という別の 2 つのクラスを宣言する必要があります。

ここでアプリケーションをリビルドして実行する場合、MainWindow は、黄色いバックグラウンドを持つリスト ボックスで埋められます。 ウィンドウ境界をドラッグして、ウィンドウをサイズ変更すると、リスト ボックスもそれに合わせてサイズ変更されます。 クライアント領域のサイズ変更は、MainWindow のスーパークラスである SDIFrame のコードによって自動的に処理 されます。.

"defineClientPane" メソッドの追加例

1. 下のコードをコピーして、Phone アプリケーション MainWindow クラス (tpltwin.cbl) の "defineClientPane" メソッドの Method-id 見出しとリンケージ節の間に貼り付けます。

```
working-storage section.

* このデータは静的に宣言されているため、working-storage で宣言されます。通常メソッドのデータは local-storage で宣言されます。

 copy "listflag.cpy".

01 tabs.
  03 filler              pic x(4) comp-5 value 160.
  03 filler              pic x(4) comp-5 value 260.
  03 filler              pic x(4) comp-5 value 360.
  03 filler              pic x(4) comp-5 value 420.
  03 filler              pic x(4) comp-5 value 600.
01 filler            redefines tabs.
```

```
  03 tab-stop             pic x(4) comp-5 occurs 5.

local-storage section.

01 tab-stops              object reference.

01 i                      pic x(4) comp-5.
```

2. 文 "invoke Subpane new..." を削除してから、下のコードをコピーして、Phone アプリケーションの MainWindow クラス (tpltwin.cbl) の "defineClientPane" メソッドにある手続き部に貼り付けます。

```
    invoke SingleSelectionListbox "new" using lnkParent
                                  returning lnkSpace
    set osList to lnkSpace
```

***----表示の設定。デフォルトでは、**`listboxes`** がテキスト行の整数になるように高さを調節します。これにより、リストボックスがウィンドウのクライアント領域に適合しないようにします。**`noAdjust`**メッセージによりこの動作をオフにします。**

```
    invoke osList "noAdjust"
    invoke osList "yellow"
```

***----タブの停止を設定する**

```
    invoke osList "addFlags" using by value lbf-use-tabstops
    move 5 to i
    invoke Array "ofValues" using cobolcomp5 i
       returning tab-stops
    perform varying i from 1 by 1 until i > 5
       invoke tab-stops "atPut" using i tab-stop(i)
     end-perform
    invoke osList "setTabStops" using tab-stops
    invoke tab-stops "finalize" returning tab-stops
```

***----リストボックス イベントを設定する**

```
    map event p2ce-accept upon osList
```

```
                to method z"onEventEdit" upon self

        map event p2ce-select upon osList

                to method z"onEventRecordSelect" upon self
```

3. Class-Control 段落に進んで、宣言を追加します。

```
        Array is class "array"

        SingleSelectionListBox is class "slistbox"
```

4. このメソッドは、Object-Storage のクライアント領域オブジェクトへのオブジェクト ハンドルも保存します。 Object-Storage 節は、このオブジェクトのメソッドすべてに使用可能なデータを宣言するために使用されます。 Object-Storage 節を探して (Class-Control 段落の少し下にある)、次の宣言を追加します。

```
    01 osList           object reference.
```

9.7 ファイルのオープンと保存

9.7.1 ファイルとメソッド

この節では、エンドユーザが phonebook ファイルを開いたり保存できるようにするコードを追加します。次の表は、[ファイル] メニューに表示される項目と、これらの項目にマッピングされているメソッドを示しています。

項目	メソッド	説明
Open master	"onEventOpenMaster"	openmaster-signal を送信する
Open user	"onEventOpen"	Windows の [ファイルを開く] ダイアログを表示し、ファイル名とパスを抽出して、fileopen シグナルを送信します。MainWindow クラスには、デフォルトの "onEventOpen" メソッドが含まれます。
インポート	"onEventImport"	Windows の [ファイルを開く] ダイアログを表示し、ファイル名とパスを抽出して fileimport シグナルを送信します。
保存	"onEventSave"	filesave シグナルを送信します。MainWindow クラスには、デフォルトの "onEventSave" メソッドが含まれます。
名前を付けて保存	"onEventSaveAs"	Windows の [ファイルを開く] ダイアログを表示し、ファイル名とパスを抽出して filesaveas シグナルを送信します。

電話帳が開かれていない場合、またはマスター電話帳 が開かれている場合にエンドユーザによる保存を禁止する場合でも、提供されている "onEventSave" メソッドを変更する必要はありません。 "onEventSave" に対して、保存が

許可されているかどうかをチェックするコードを追加する代わりに、[保存] メニュー項目を使用不能にして、マスター電話帳以外のファイルが正常に開かれている場合にのみこの項目を使用可能にするコードを追加できます。

提供されている "onEventOpen" メソッドは、必要な機能のほとんどをすでに実行しています。ただし、[ファイルを開く] ダイアログ　ボックスに表示されるファイルを、拡張子 *.lst を持つものに限定する必要があります。[ファイルの上書き保存]ダイアログ ボックスでも同じことを実行する必要があるため、"setFileFilter" と呼ばれるメソッドに、ファイル ダイアログ ボックスへのファイル フィルタを設定するためのすべてのコードを挿入して、ファイル フィルタを設定するたびにこのコードを起動します。

9.7.2 "setFileFilter" メソッドの説明

このメソッドは、[ファイルを開く]または[ファイルの上書き保存] ダイアログ ボックスをパラメータとしてとり、それを "addFilter" メッセージに送信します。最初のパラメータは、ダイアログ ボックスにある型フィールドのファイルの説明を提供します。2 番めのパラメータは、ダイアログ ボックスが型 .lst のファイルだけを表示するように制限するフィルタです。

どちらのパラメータも CharacterArrays であり、メソッドの最後に最終化されます。FileDialog クラスは、渡されたデータの独自のコピーをとるため、これらのオブジェクトを廃棄して、メモリ リークを避ける必要があります。 すべてのクラス ライブラリ オブジェクトは、Collection クラスを除いて、パラメー タとして渡されたオブジェクトのコピーをとります。

"setFileFilter" メソッドの追加例

1. 下のコードをコピーし、Phone アプリケーションの MainWindow クラス (tpltwin.cbl)の、END-OBJECT コード (プログラムの終わり近く) の直前に貼り付けます。

```
*>------------------------------------------------------------------
*> Phonebook ファイル (拡張子 .ls) を表示するために FileSave または FileOpen
*> ダイアログ ボックス上でファイル フィルタを設定する
*>------------------------------------------------------------------
 method-id. "setFileFilter".
 local-storage section.
 01 lsString              object reference.
 01 lsString2             object reference.
 linkage section.
 01 lnkFileDialog          object reference.
```

```
        procedure division using lnkFileDialog.


            invoke CharacterArray "withValue"
                                using z"Phonebooks (*.lst)"
                                        returning lsString
            invoke CharacterArray "withValue" using z"*.lst"
                                    returning lsString2
            invoke lnkFileDialog "addFilter" using lsString lsString2
            invoke lsString "finalize" returning lsSTring
            invoke lsString2 "finalize" returning lsSTring2
            exit method.
        end method "setFileFilter".
```

2. MainWindow クラスの "onEventOpen" メソッドを検索して、[ファイルを開く] ダイアログを表示する文 (invoke lsOpenDialog "show") の直前に、下の文を入力します。

```
invoke self "setFileFilter" using lsOpenDialog
```

9.7.3 "onEventImport" メソッドの説明

このメソッドは、"onEventOpen" に似ていますが、異なるシグナルを送信して、[ファイルを開く]ダイアログ ボックスに異なるタイトルを設定します。

"onEventImport" メソッドの追加例

下のコードをコピーし、Phone アプリケーションの MainWindow クラス (tpltwin.cbl)の、END-OBJECT コード (プログラムの終わり近く) の直前に貼り付けます。

```
      method-id. "onEventImport".


      local-storage section.
      01 lsDataBlock           object reference.
```

```
 01 lsOpenDialog           object reference.
 01 lsString               object reference.
 01 lsMessage              object reference.
 procedure division.
*----[ファイルを開く] ダイアログを表示する
    invoke OpenDialog "new" using self
                    returning lsOpenDialog
    invoke self "setFileFilter" using lsOpenDialog
    invoke lsOpenDialog "setTitleZ" using z"Import phonebook"
    invoke lsOpenDialog "show"
*----ダイアログから情報を抽出する
    invoke lsOpenDialog "getFile" returning lsString
    if lsString <> null
       invoke SymbolTable "new" returning lsDataBlock
       put value lsString into table lsDataBlock
             at symbol filepath-key
       invoke lsOpenDialog "getFileTitle" returning lsString
       put value lsString table lsDataBlock symbol filename-key
*--------[ファイルを開く] ダイアログを完成させる
       invoke lsOpenDialog "finalize" returning lsOpenDialog
       invoke self "update"
*--------オープンシグナルを送る
       invoke wsEventManager "setSystemBusy"
       signal import-signal lsDataBlock
*---------クリーンアップする
```

```
            invoke lsDataBlock "deepFinalize" returning lsDataBlock
        end-if
        exit method.
    end method "onEventImport".
```

9.7.4 "onEventOpenMaster" メソッドの説明

このメソッドは、ファイルを開いたりインポートするためのメソッドよりもはるかに簡単です。このシグナルの受信側は、マスターファイルのファイル名を判断して、このメソッドが openmaster シグナルだけを送信するようにする必要があります。

ここまでに追加したメソッドは、どれも着信データを処理しません。シグナルを送信するだけです。 これらのシグナルを受信するコンポーネントは、成功または失敗を示すか、または表示するレコード セットを送信する他のシグナルを介して結果を返します。

"onEventOpenMaster" メソッドの追加例

下のコードをコピーし、Phone アプリケーションの MainWindow クラス (tpltwin.cbl)の、END-OBJECT コード (プログラムの終わり近く) の直前に貼り付けます。

```
*>------------------------------------------------------
*> ウィンドウ イベント メソッド
*> 選択中のファイルで起動し、マスターを開く
*>------------------------------------------------------
 method-id. "onEventOpenMaster".
 local-storage section.
 01 lsDataBlock           object reference.

 procedure division.
        invoke SymbolTable "new" returning lsDataBlock
        invoke wsEventManager "setSystemBusy"
         signal openmaster-signal lsDataBlock
```

```
            invoke lsDataBlock "deepFinalize" returning lsDataBlock
            exit method.
        end method "onEventOpenMaster".
```

9.7.5 "onEventSaveAs" メソッドの説明

このメソッドは、次のことを行います。

本来の [ファイルの上書き保存] ダイアログ ボックスを表示する シンボル テーブルにファイルとパス情報を挿入する saveas シグナルを送信する

これで、ファイルを開いて保存するためのインタフェース コードの追加が終わりました

"onEventSaveAs" メソッドの追加例

下のコードをコピーし、Phone アプリケーションの MainWindow クラス (tpltwin.cbl)の、END-OBJECT コード (プログラムの終わり近く) の直前に貼り付けます。

```
        *>------------------------------------------------------
        *> ウィンドウ イベント メソッド
        *> 選択中のファイルで起動し、Raises としてファイルを保存する
        *> signal saveas-signal.
        *>------------------------------------------------------
         method-id. "onEventSaveAs".
         local-storage section.
         01 lsDataBlock           object reference.
         01 lsSaveDialog          object reference.
         01 lsString              object reference.
         01 lsMessage             object reference.
         procedure division.
        *----[ファイルを保存] ダイアログを表示する
```

```
     invoke SaveDialog "new" using self
                    returning lsSaveDialog
    invoke self "setFileFilter" using lsSaveDialog
    invoke lsSaveDialog "show"
*----ダイアログから情報を抽出する
    invoke lsSaveDialog "getFile" returning lsString
    if lsString <> null
       invoke SymbolTable "new" returning lsDataBlock
       put value lsString into table lsDataBlock
             at symbol filepath-key
       invoke lsSaveDialog "getFileTitle" returning lsString
       put value lsString table lsDataBlock symbol filename-key
*--------[ファイルを保存] ダイアログを完成させる
       invoke lsSaveDialog "finalize" returning lsSaveDialog
       invoke self "update"
       invoke wsEventManager "setSystemBusy"
       if lsDataBlock <> null
*--------シグナルとして save を送る
       signal saveas-signal lsDataBlock
*---------クリーンアップする
       invoke lsDataBlock "deepFinalize" returning lsDataBlock
    else
       invoke lsSaveDialog "finalize" returning lsSaveDialog
    end-if
    exit method.
```

```
end method "onEventSaveAs".
```

2. このメソッドは、新しいクラス SaveDialog を使用します。次の宣言を MainWindow class-Control 段落に追加します。

```
SaveDialog is class "savedlg"
```

9.8 レコードの追加、編集、および削除

9.8.1 ファイルとメソッド

この節では、エンドユーザがレコードの追加、編集、および削除を実行できるようにするコードを追加します。 次の表は、[レコード] メニューに表示される項目と、これらの項目にマッピングされているメソッドを示しています。

項目	メソッド	説明
追加	"onEventAdd"	[レコードの詳細] ダイアログ ボックスを表示します。dialog-OK イベント を "onDialogAddOk" にマッピングし、dialog-cancel メソッドを "onDialogCancel" にマッピングします。
編集	"onEventEdit"	[レコードの詳細] ダイアログを表示して、現在選択されているレコードの内容を設定します。dialog-OK イベントを "onDialogEditOk" にマッピングし、dialog-cancel メソッドを "onDialogCancel" にマッピングします。
削除	"onEventDelete"	確認メッセージ ボックスを表示します。エンドユーザが [OK] をクリックすると、現在選択されているレコードが削除されます。

9.8.2 "onEventAdd" メソッドの説明

クライアント領域リスト ボックスで現在選択されているレコードをフェッチし、これを現在選択されているレコードとして (osCurrentRecord に) 保存します。 フィールドをブランクにしたまま、[レコードの詳細]ダイアログ ボックスを表示します。 dialog-ok イベントをメソッド "onEventAddOk" にマッピングし、dialog-cancel イベントをメソッド "onDialogCancel" にマッピングします。 "onEventAddOk" メソッドは、エンドユーザがダイアログ ボックスで [OK] をクリックして、新しいレコードを追加するためのシグナルを送信すると起動されます。

"onEventAdd" メソッドの追加例

1. 下のコードをコピーし、Phone アプリケーションの MainWindow クラス (tpltwin.cbl)の、END-OBJECT コード (プログラムの終わり近く) の直前に貼り付けます。

```
*>------------------------------------------------------------------

*> 選択中の Record で起動し、追加された ウィンドウ イベント メソッド
```

```
*> 新規レコードを追加するためにダイアログを表示する
*>-----------------------------------------------------------------
 method-id. "onEventAdd".
 working-storage section.
 01 wsBlanks.
  03 filler                pic x(79) value spaces.
  03 filler                pic x(1) value x"00".
 local-storage section.
 01 lsString               object reference.
 01 lsRecord               object reference.
 01 lsDataBlock             object reference.
 01 lsDialog               object reference.
 01 lsId                   pic x(4) comp-5.

 procedure division.
*----リスト ボックス内で選択したレコードの値を取り出す
    invoke osList "selectedString" returning lsString
    if lsString = null
*--------選択されたレコードがないため、全スペースを使って文字列を作成する
        invoke CharacterArray "withValue" using wsBlanks
                                  returning lsString
    end-if
*----選択した文字列にある値から TelRecord を作成する
    invoke TelRecord "withAll" using lsString
                         returning lsRecord
```

```
*----インスタンス データにある選択したレコードの値を記録する
*    新規レコードに位置付けるためにアプリケーション要素により選択レコードが使用
     され
     set osCurrentRecord to lsRecord
     invoke lsString "finalize" returning lsString
*----[Record Details(レコードの詳細)] ダイアログ ボックスを作成
     move IDD-RECORD-DETAILS to lsId
     invoke RecordDetailsDialog "fromResource"
                                using self osResource lsId
                              returning lsDialog
*----OK にマップし、イベントをキャンセルする
     map event dialog-ok upon lsDialog
               to method z"onDialogAddOk" upon self
     map event dialog-cancel upon lsDialog
               to method z"onDialogCancel" upon self
*----表示
     invoke lsDialog "show"
     exit method.
 end method "onEventAdd".
```

2. Class-Control 段落に進んで、次の宣言を追加します。

```
     TelRecord is class "telrec"
     RecordDetailsDialog is class "recrdbox"
```

3. インスタンス データの新しい項目を宣言して、現在選択されているレコードの値を保持する必要もあります。MainWindow Object-Storage 節を検索して、次の宣言を追加します。

```
 01 osCurrentRecord     object reference.
```

9.8.3 "onDialogAddOK" メソッドの説明

"onDialogAddOK" メソッドは、次のことを行います。

- メソッドに渡されたイベント オブジェクトから、ダイアログ ボックスのオブジェクト ハンドルを取得します。
- ダイアログ ボックスの内容を取得します。
- ダイアログ ボックスと osCurrentRecord の内容を SymbolTable に挿入し、add シグナルを送信します。
- SymbolTable を最終化して、メモリ リークを防止します。これは、osCurrentRecord に保存されたオブジェクトも最終化します。 あるいは、次に "onEventAdd" または "onEventEdit" が起動されたときに、新しいオブジェクト ハンドルが osCurrentRecord に保存され、すでにそこにあったオブジェクト ハンドルは失われます。 アプリケーションがこのオブジェクト ハンドルを介してこのオブジェクトにアクセスする権利がなくなりますが、オブジェクト自体は引き続き存在しています。
- osCurrentRecord の値をヌルに設定します。"deepFinalize" が、osCurrentRecord によって参照されるオブジェクトを廃棄しましたが、オブジェクト ハンドルは引き続き osCurrentRecord に存在します。 値をヌルに設定すると、他のコードで、osCurrentRecord の内容をヌルと比較することによって、それがオブジェクトを指しているかどうかをテストできます。

[追加] または [編集] ダイアログ ボックスがキャンセルされた場合は、必ず MainWindow でも osCurrentRecord を最終化する必要があります。 これらのダイアログはいずれも、"onDialogCancel" によって廃棄されるため、いくつかの追加コードが必要になります。

"onDialogAddOK" メソッドの追加例

下のコードをコピーし、Phone アプリケーションの MainWindow クラス (tpltwin.cbl)の、END-OBJECT コード (プログラムの終わり近く) の直前に貼り付けます。

```
*>-----------------------------------------------------------------
*> [Add record] ダイアログで OK ボタンが押された時に起動された
*> ウィンドウ イベント メソッド。新規レコードを追加する
*>-----------------------------------------------------------------
 method-id. "onDialogAddOK".

 local-storage section.
```

```
01 lsDialog              object reference.
01 lsString              object reference.
01 lsDataBlock           object reference.
linkage section.
01 lnkEvent              object reference.

procedure division using lnkEvent.
   invoke lnkEvent "getObject" returning lsDialog
   invoke lsDialog "hide"
      invoke SymbolTable "new" returning lsDataBlock
   invoke lsDialog "getFields" returning lsString
      put value lsString into table lsDataBlock
                        at symbol record-key
   put value osCurrentRecord into table lsDataBlock
                     at symbol position-key
   invoke lsDialog "finalize" returning lsDialog
   signal add-signal lsDataBlock
      invoke lsDataBlock "deepFinalize" returning lsDataBlock
   set osCurrentRecord to null
   exit method.
end method "onDialogAddOK".
```

9.8.4 追加のダイアログ キャンセル コードの説明

追加したばかりの新しいコードは、osCurrentRecord にヌルが含まれるかどうかをテストします。ヌルが含まれない場合は、最終化が必要なオブジェクトが存在します。"finalize" メソッドは、ヌルをパラメータとして返し、これは osCurrentRecord に挿入されます。

onDialogCancel への新しいコードの追加例

1. MainWindow クラスの "onDialogCancel" メソッドを検索します。
2. 次のコードを、EXIT METHOD 文の直前に追加します。

```
if osCurrentRecord <> null
   invoke osCurrentRecord "finalize"
                          returning osCurrentRecord
end-if
```

9.8.5 "onEventEdit" メソッドの説明

このコードは、[レコードの詳細] ダイアログボックスを表示して、そのフィールドに現在選択されているレコードの内容を設定します。これは、dialog-ok イベントをメソッド "onEventEditOk" に、dialog-cancel イベントをメソッド "onDialogCancel" にマッピングもします。

"onEventEdit" メソッドの追加例

下のコードをコピーし、Phone アプリケーションの MainWindow クラス (tpltwin.cbl)の、END-OBJECT コード (プログラムの終わり近く) の直前に貼り付けます。

```
*>------------------------------------------------------------------
*> 選択中の Record で起動され、編集されたウィンドウ イベント メソッド
*> 新規レコードを編集するためにダイアログを表示する
*>------------------------------------------------------------------
 method-id. "onEventEdit".
 local-storage section.
 01 lsString              object reference.
 01 lsMessageBox          object reference.
 01 lsRecord              object reference.
 01 lsDialog              object reference.
 01 lsId                  pic x(4) comp-5.
```

```
 procedure division.
*----Fetch the currently selected record, and create a telephone
*    record
     invoke osList "selectedString" returning lsString
     invoke TelRecord "withAll" using lsString
                          returning lsRecord
*----Store the currently selected record in instance data
     set osCurrentRecord to lsRecord
     invoke lsString "finalize" returning lsString
*----Create a record details dialog box
     move IDD-RECORD-DETAILS to lsId
     invoke RecordDetailsDialog "fromResource"
                               using self osResource lsId
                            returning lsDialog
*----Map the dialog boxes to methods
     map event dialog-ok upon lsDialog
               to method z"onDialogEditOk" upon self
     map event dialog-cancel upon lsDialog
               to method z"onDialogCancel" upon self
*----Send the dialog box the "create" method - this sets up the
*    object handles for the entry fields on the dialog box.
     invoke lsDialog "create"
*----Set the data on the dialog box, and display it.
     invoke lsDialog "setFields" using lsRecord
```

```
           invoke lsDialog "show"

           exit method.

       end method "onEventEdit".
```

9.8.6 "onDialogEditOK" メソッドの説明

このメソッドは、ダイアログ ボックスの内容を抽出して、それを SymbolTable に挿入し、recordEdit シグナルとともに送信します。"onDialogAddOk" と同様、このメソッドは、オブジェクトが終了すると、SymbolTable を最終化して、osCurrentRecord をヌルに設定します。

"onDialogEditOK" メソッドの追加例

下のコードをコピーし、Phone アプリケーションの MainWindow クラス (tpltwin.cbl)の、END-OBJECT コード (プログラムの終わり近く) の直前に貼り付けます。

```
       *>-----------------------------------------------------------------
       *> OK ボタンが [Edit record] ダイアログで押されたときに起動された
       *> ウィンドウ イベント メソッド。レコードを修正する
       *>-----------------------------------------------------------------
        method-id. "onDialogEditOK".

        local-storage section.
        01 lsDialog              object reference.
        01 lsString              object reference.
        01 lsDataBlock           object reference.
        linkage section.
        01 lnkEvent              object reference.

        procedure division using lnkEvent.
```

```
        invoke lnkEvent "getObject" returning lsDialog
        invoke lsDialog "hide"
            invoke SymbolTable "new" returning lsDataBlock
        invoke lsDialog "getFields" returning lsString
            put value lsString into table lsDataBlock
                                at symbol record-key
        put value osCurrentRecord into table lsDataBlock
                            at symbol position-key
        invoke lsDialog "finalize" returning lsDialog
        signal edit-signal lsDataBlock
            invoke lsDataBlock "deepFinalize" returning lsDataBlock
        set osCurrentRecord to null
        exit method.
    end method "onDialogEditOK".
```

9.8.7 "onEventDelete" メソッドの説明

"onEventDelete" メソッドは、次のことを行います。

メッセージ ボックスを表示して、そこに "okCancel" メソッドを送信します。 これにより、[OK] ボタンと [キャンセル] ボタンが作成されます (デフォルトのメッセージ ボックスには、[OK] ボタンだけが表示されます)。 メッセージ ボックスからの戻り値をテストして、[OK] ボタンがクリックされたかどうかを確認します。 メッセージ ボックスは、ダイアログ ボックスよりも処理が簡単です。エンドユーザがクリックしたボタンを知らせる "show" メッセージでパラメータを返します。これは、メソッドにイベントをマッピングして、異なるイベントを処理するための異なるメソッドをコーディングするよりも簡単です。 メッセージ ボックスは制限されているため、この単純なインタフェースを備えています。このインタフェースによって、ユーザは、メッセージを設定し、少数の定義済みボタン スタイルの 1 つ (たとえば "ok" と "okcancel") を選択できます。

現在選択されているレコードを SymbolTable に挿入して、delete シグナルを送信します。

現在選択されているレコードとは、クライアント領域のリスト ボックスで強調表示されているレコードをいいます。現在レコードが選択されていない場合は、編集および削除の各機能を使用不能にする必要があります。この処置は、メソッド "disableEditFunctions" を作成して、"initialize" メソッドからこのメソッドを呼び出して行います。 リスト

ボックスに新しいレコード セットが表示されると、選択を示す強調表示が削除されるため、この場合も必ずこのメソッドを起動します。

"onEventDelete" メソッドの追加例

1. 下のコードをコピーし、Phone アプリケーションの MainWindow クラス (tpltwin.cbl)の、END-OBJECT コード (プログラムの終わり近く) の直前に貼り付けます。

```
*>------------------------------------------------------------------
*> 選択中の Record で起動され、削除されたウィンドウ イベント メソッド
*> レコードを削除する
*>------------------------------------------------------------------
 method-id. "onEventDelete".
 local-storage section.
 01 lsString              object reference.
 01 lsMessageBox           object reference.
 01 lsRecord              object reference.
 01 lsDialog              object reference.
 01 lsDataBlock           object reference.
 01 lsId                  pic x(4) comp-5.

 procedure division.
    invoke osList "selectedString" returning lsString
    invoke MessageBox "new" using self returning lsMessageBox
    invoke lsMessageBox "setTitleZ" using wsWindowTitle
*----確認メッセージ ボックスを表示する
    invoke lsMessageBox "okcancel"
    invoke lsMessageBox "setMessageZ"
```

```
                                using wsMsgDeleteRecord
        invoke lsMessageBox "show" returning lsId
        invoke lsMessageBox "finalize" returning lsMessageBox
        if lsId = mb-return-ok
            invoke SymbolTable "new" returning lsDataBlock
    *--------現在選択したレコードの値から電話番号レコードを作成する
        invoke TelRecord "withAll" using lsString
                            returning lsRecord
        invoke lsString "finalize" returning lsString
    *--------シンボル表にそれを格納する
            put value lsRecord into table lsDataBlock
                            at symbol record-key
            invoke wsEventManager "setSystemBusy"
    *--------削除シグナルを送る
            signal delete-signal lsDataBlock
            invoke lsDataBlock "deepFinalize"
                        returning lsDataBlock
        end-if
        exit method.
     end method "onEventDelete".
```

2. このメソッドは、ユーザに対して、メッセージ ボックスの削除を確認するようにプロンプトで要求します。メッセージ テキストを MainWindow Working-Storage に追加する必要があります。MainWindow Working-Storage 節 (プログラムの一番上近く、Class-Control 段落の下) を検索してください。

3. 次の 03 レベルの宣言を wsMessages に追加します。

```
           03 wsMsgDeleteRecord    pic x(25)
                                  value z"このレコードを削除しますか?".
```

9.8.8 "disableEditFunctions" メソッドの説明

このメソッドは、"getObjectFromId" メソッドを使用して、[編集] および [削除] の各メニュー項目 (シンボル ID-RECORD-EDIT および ID-RECORD-DELETE によって表わされる) をフェッチし、それらに "disable" メッセージを送信します。メニュー項目に "disable" を送信すると、項目はグレー表示されるため、エンドユーザはこの項目を選択できません。関連のツールバー ボタンがある場合も、同様に使用不能になります。

リスト ボックスに定義するコードを追加したときに、ユーザがレコードを選択すると必ず発生するイベントを、メソッド "onEventRecordSelect" にマッピングしました。このメソッドを使用すると、これらの機能を実行できます。

"disableEditFunctions" メソッドの追加例

下のコードをコピーして、END-OBJECT コード (プログラムの終わり近く) の直前にある Phone アプリケーション MainWindow クラス (tpltwin.cbl) に貼り付けます。

```
*>-----------------------------------------------------------------
*> レコードの［編集］と［削除］機能を使用不可にする
*> リスト ボックスに選択されたレコードがない場合に起動される
*>-----------------------------------------------------------------
 method-id. "disableEditFunctions".
 local-storage section.
 01 lsItem               object reference.
 01 lsId                 pic x(4) comp-5.
 procedure division.
    move ID-RECORD-EDIT to lsId
    invoke self "getObjectFromId" using lsId returning lsItem
    invoke lsItem "disable"
    move ID-RECORD-DELETE to lsId
    invoke self "getObjectFromId" using lsId returning lsItem
```

```
        invoke lsItem "disable"

        exit method.

    end method "disableEditFunctions".
```

9.8.9 "onEventRecordSelect" メソッドの説明

このメソッドは、[編集] および [削除] の各メニュー項目を使用可能にし、"disableEditFunctions" と同様に動作します。

最終的に、追加機能は、phonebook が開かれるまで使用不能になります。このコードは、"disableEditFunctions" を起動する文とともに、"initialize" メソッドに追加します。

"onEventRecordSelect" メソッドの追加例

下のコードをコピーし、Phone アプリケーションの MainWindow クラス (tpltwin.cbl)の、END-OBJECT コード (プログラムの終わり近く) の直前に貼り付けます。

```
*>------------------------------------------------------
*> ウィンドウ イベント メソッド
*> リスト ボックス内に選択中のレコードで起動
*> [編集] と [削除] 機能を使用可能にする
*>------------------------------------------------------
 method-id. "onEventRecordSelect".
 01 lsId                 pic x(4) comp-5.
 01 lsItem               object reference.
 procedure division.
    move ID-RECORD-EDIT to lsId
    invoke self "getObjectFromId" using lsId returning lsItem
     invoke lsItem "enable"
    move ID-RECORD-DELETE to lsId
    invoke self "getObjectFromId" using lsId returning lsItem
```

```
            invoke lsItem "enable"

            exit method.

        end method "onEventRecordSelect".
```

9.8.10 "initialize" コードの説明

このコードも、使用不能にするメニュー項目を取得して、それらに "disable" メッセージを送信します。

"initialize" コードの追加例

1. 下のコードをコピーして、Phone アプリケーション MainWindow クラス (tpltwin.cbl) の "Initialize" メソッドの "ユーザコード" と "ユーザ終り" コメントの間に貼り付けます。

```
            *----はじめに使用可能でなかったメニュー機能を使用不可にする

                invoke self "disableEditFunctions"

                move ID-FILE-SAVE to lsId

                invoke self "getObjectFromId" using lsId

                                    returning lsItem

                invoke lsItem "disable"

                move ID-RECORD-ADD to lsId

                invoke self "getObjectFromId" using lsId

                                    returning lsItem

                invoke lsItem "disable"
```

2. この新しいコードは、まだ宣言されていない 2 つのデータ項目 lsId と lsItem を使用します。Local-Storage 節を "initialize" メソッドに追加する必要があります。メソッドの一番上にある Linkage-Section のすぐ上に、次のコードを "initialize" メソッドに 入力します。

```
        local-storage section.

        01 lsId                  pic x(4) comp-5.

        01 lsItem                object reference.
```

9.9 ソート機能の追加

まだ追加していないユーザ インタフェース機能は、ソート機能だけです。[ファイル] メニューの [Sort] は、"onEventSort" メソッドにマッピングされています。

9.9.1 "onEventSort" メソッドの説明

このメソッドは、sort シグナルを送信します。シグナルの受信側は、ソートを実行して、ソートされたデータを返します。

"onEventSort" メソッドの追加例

下のコードをコピーし、Phone アプリケーションの MainWindow クラス (tpltwin.cbl)の、END-OBJECT コード (プログラムの終わり近く) の直前に貼り付けます。

```
*>------------------------------------------------------
*> ウィンドウ イベント メソッド
*> 選択中の File で起動され、ソートさせる
*>------------------------------------------------------
 method-id. "onEventSort".
 procedure division.
    invoke wsEventManager "setSystemBusy"
    signal sort-signal
    exit method.
 end method "onEventSort".
```

9.10 シグナルへの応答

この時点で、エンドユーザの要求に応じる MainWindow クラスは取得されていますが、シグナルを受信または処理する方法はありません。 この節では、このコンポーネントへの入力シグナルを、これらのシグナルを処理するコードにマッピングします。

9.10.1 シグナルマップの機能

次に示すのは別のコードであり、MAP SIGNAL 文に定義されたボキャブラリを使用します。

```
map signal displaydata-signal from mainApplicationSocket
                        to method z"onSignalDisplayData"
```

上記のコードは、次のコードと等価です。

```
invoke self "defineInSignal" using mainApplicationSocket
                              displaydata-signal z"onSignalDisplayData"
```

各 MAP SIGNAL 文は、特定のシグナルを受信したときにどのメソッドを起動するかを指示します。mainApplicationSocket パラメータは、シグナルが表示されるソケットをを示します。コンポーネントでは、入力シグナルを受信する複数のソケットを定義できます。 このアプリケーションの MainWindow クラスは、mainApplicationSocket だけを使用します。 このことは、次の文によって、このメソッドの最初に宣言されています。

```
declare socket mainApplicationSocket
```

これはもう 1 つのボキャブラリ定義文です。

次の表は、入力シグナルと、それらのマッピングされたメソッドを示しています。

シグナル	メソッド	説明
fileOpen-signal	"onSignalFileOpen"	ウィンドウのタイトルバーを変更してファイルの名前を含めて、ユーザ電話帳またはマスター電話帳のどちらが開かれたかによって、[保存] メニュー項目を使用可能または使用不能にします。テンプレートにはデフォルト メソッドが組み込まれていますが、機能は何も含まれていません。
fileOpenFailed-signal	"onSignalOpenFailed"	オープンが正常に実行されなかったことを知らせるメッセージを表示します。テンプレートにデフォルト メソッドが含まれています。
searchFailed-signal	"onSignalSearchFailed"	一致が検出されなかったことを知らせるメッセージ ボックスを表示します。
displayData-signal	"onSignalDisplayData"	シグナルとともに送信されたすべてのレコードを、クライアント領域のリストボックスに表示します。
saved-signal	"onSignalFileOpen"	ウィンドウのタイトルバーを変更して、ファイルの名前を含めます (このシグナルが [名前を付けて保存] が原因で送信された場合、ファイル名は変更されています)。

savefailed-signal	"onSignalSaveFailed	保存が正常に実行されなかったことを知らせるメッセージを表示します。

テンプレートに含まれるデフォルト メソッドには必要な機能すべてがあるため、ここで、"onSignalOpenFailed" を除く、上記メソッドすべてのコードを追加する必要があります。

この時点で、ウィンドウのタイトルを "Window" から "Phonebook" に変更することをお勧めします。 MainWindow クラスの Working-Storage 節にで、wsWindowTitle の宣言を検索してください。これを次のように変更します。

```
03  wsWindowtitle          pic x(7) value z"電話帳".
```

新しいシグナルの定義例

下のコードをコピーして、Phone アプリケーション MainWindow クラス (tpltwin.cbl) の "defineInputSockets" メソッドの "ユーザコード" と "ユーザ終り" コメントの間に貼り付けます。

```
map signal searchfailed-signal from mainApplicationSocket
                    to method z"onSignalSearchFailed"
map signal displaydata-signal from mainApplicationSocket
              to method z"onSignalDisplayData"
map signal saved-signal from mainApplicationSocket
                       to method z"onSignalFileOpen"
map signal savefailed-signal from mainApplicationSocket
                       to method z"onSignalSaveFailed"
```

9.10.2 "onSignalFileOpen" の説明

このメソッドは、次の 2 つのことを実行します。

開かれているファイルの名前によって、ウィンドウのタイトルを更新する。 このファイルを更新できない場合に [保存] メニュー項目を使用不能にし、ファイルを更新できる場合に項目を使用可能にする。 シグナル lnkDataBlock とともに送信されたデータは SymbolTable です。

このメソッドを詳しく調べると、ボキャブラリ定義文句 FETCH VALUE を使用して、シグナルとともに送信されたデータのオブジェクト lnkDataBlock に問い合わせていることがわかります。

次の文を参照してください。

```
fetch value lsFilename from table lnkDataBlock
                    at symbol filepath-key
```

この文は、filepath-key に対して保存されたオブジェクトを lsFilename に返します。規則により、SymbolTables に保存されたオブジェクトは、CharacterArrays または OrderedCollections のどちらかです。 この場合は、CharacterArray です。ファイル名を含む CharacterArray を抽出すると、その値はバッファに読み込まれて、ウィンドウの新しいタイトルを作成するために使用されます。 新しいタイトルは、MainWindow によって、それ自体にメッセージ "setTitleZ" を送信することで設定されます。

```
invoke self "setTitleZ" using lsWindowTitle
```

コードの次の節は、シンボル ステータスキーに保存された CharacterArray をフェッチし、その値を StatusUpdate (78 レベルデータ名) と比較します。 この結果に従って、[保存] メニュー項目は使用可能または使用不能になります。

メソッド "onSignalFileOpen" は、実際にはデータを何も表示しません。displayData シグナルによって起動されるメソッド "onSignalDisplayData" がデータを表示します。

"onSignalFileOpen" メソッドの追加例

1. MainWindow クラスの "onSignalFileOpen" メソッドを検索して、削除します。
2. 下のコードをコピーし、Phone アプリケーションの MainWindow クラス (tpltwin.cbl)の、END-OBJECT コード (プログラムの終わり近く) の直前に貼り付けます。

```
*>-----------------------------------------------------------------
*> アプリケーションが正しくファイルを開いた場合はいつでも起動される
*> ウィンドウシグナルメソッド
*>-----------------------------------------------------------------
 method-id. "onSignalFileOpen".

 local-storage section.
 01 lsFilename            object reference.
 01 lsStatus              object reference.
 01 lsWindowTitle         pic x(100).
```

```
01 lsString              pic x(100).
01 lsId                  pic x(4) comp-5.
01 lsLength              pic x(4) comp-5.
01 lsMenuItem             object reference.

linkage section.
01 lnkDataBlock           object reference.

procedure division using lnkDataBlock.
*----ユーザコード．正しくファイルを開くためのコードを追加する
*----ウィンドウのタイトルを更新する
    fetch value lsFilename from table lnkDataBlock
             at symbol filepath-key
    move length of lsString to lsLength
    invoke lsFilename "getValueWithSize" using lsLength
                                returning lsString
    move wsWindowTitle to lsWindowTitle
    move " - " to lsWindowTitle(10:)
    move lsString to lsWindowTitle(13:)
    perform varying lsLength from length of lsWindowTitle
           by -1 until lsWindowTitle(lsLength:1) not = space
           or lsLength = 1
     end-perform
        add 1 to lsLength
    move x"00" to lsWindowTitle(lsLength:1)
```

```
            invoke self "setTitleZ" using lsWindowTitle
        *----[保存] メニュー項目を非表示にするか決定する
            fetch value lsStatus from table lnkDataBlock
                                at symbol status-key
            move spaces to lsString
            invoke lsStatus "getValueZ" returning lsString
            move ID-FILE-SAVE to lsId
            invoke self "GetObjectFromId" using lsId
                                        returning lsMenuItem
            if lsString <> StatusUpdate
               invoke lsMenuItem "disable"
            else
               invoke lsMenuItem "enable"
            end-if
            move ID-RECORD-ADD to lsId
            invoke self "GetObjectFromId" using lsId
                                        returning lsMenuItem
            invoke lsMenuItem "enable"
        *----ユーザ終り.
            exit method.
         end method "onSignalFileOpen".
```

9.10.3 "onSignalDisplayData" の説明

このメソッドは、3 つの文だけからなる非常に簡単なものです。最初の文は、シグナルとともに送信された SymbolTable から、表示するデータを抽出します。

```
fetch value lsCollection from table lnkDataBlock
                    at symbol collectionref-key
```

今回、送信されたデータは、CharacterArrays を含む OrderedCollection です。各 CharacterArray は、単一の phonebook レコードを示すデータです。

次の文は、Phonebook ウィンドウのクライアント領域にこのデータを表示します。

```
invoke osList "setContents" using lsCollection
```

クライアント領域はリスト ボックスであり、ここで "setContents" を使用して、コレクションの内容を表示するように設定できます。

メソッドの最後の文は、メソッド "disableEditFunctions" を起動します。これは、このチュートリアルの最初に MainWindow クラスに追加したものです。この文は、[Record] メニューの [Edit record] および [Delete record] の各メニューを使用不能にします。"setContents" をリスト ボックスに送信すると、以前に強調表示された選択はすべて選択解除されます。

この時点で、Phonebook アプリケーションをリビルドして実行すると、ほとんどの機能が備わっています。電話帳を開いて、レコードを追加、削除、編集できます。 また、検索も実行できますが、一致がない場合でも通知されません。アプリケーションがファイルを保存できない場合にも通知されません。 Phonebook [ファイル] メニューの [終了]をクリックしてから、このチュートリアルを終了してください。

検索失敗の通知は、"onSignalSearchFailed" メソッドによって行われます。

"onSignalDisplayData" メソッドの追加例

下のコードをコピーし、Phone アプリケーションの MainWindow クラス (tpltwin.cbl)の、END-OBJECT コード (プログラムの終わり近く) の直前に貼り付けます。

```
*>------------------------------------------------------
*> ウィンドウシグナルメソッド
*> このメソッドはアプリケーションが表示するレコードの
*> 最新リストを持っているときに起動されます
*>------------------------------------------------------
 method-id. "onSignalDisplayData".

 local-storage section.
```

```
01 lsCollection          object reference.
01 lsStatus              pic x(9).
01 lsId                  pic x(4) comp-5.
01 lsItem                object reference.

linkage section.
01 lnkDataBlock          object reference.
procedure division using lnkDataBlock.
*----データのコレクションでリストボックスを作成する
    fetch value lsCollection from table lnkDataBlock
                at symbol collectionref-key
    invoke osList "setContents" using lsCollection
*----次の "setContents" ではレコードが選択されていないため、
*    レコードの［編集］と［削除］機能を使用不可にする必要がある
    invoke self "disableEditFunctions"
*----ユーザ終り.
    exit method.
end method "onSignalDisplayData".
```

9.10.4 "onSignalSearchFailed" の説明

このメソッドは、シグナルによって渡されたデータをまったく使用しません。MessageBox を作成して、それをエンドユーザに表示するだけです。

"onSignalSearchFailed" メソッドの追加例

1. 下のコードをコピーし、Phone アプリケーションの MainWindow クラス (tpltwin.cbl)の、END-OBJECT コード (プログラムの終わり近く) の直前に貼り付けます。

```
*>-------------------------------------------------
*> ウィンドウシグナルメソッド
*> このメソッドはウィンドウが Searchfailed シグナルを受け取ったときに
*> いつでも起動され、要素が検索文字列の検索をしようとしたが失敗したことを表す
*>-------------------------------------------------
 method-id. "onSignalSearchFailed".
 local-storage section.
 01 lsMsgBox              object reference.
 linkage section.
 01 lnkDataBlock           object reference.

 procedure division using lnkDataBlock.
    invoke MessageBox "new" using self returning lsMsgBox
    invoke lsMsgBox "setTitleZ" using wsWindowtitle
    invoke lsMsgBox "setMessageZ" using wsMsgSearchFailed
    invoke lsMsgBox "show"
    invoke lsMsgBox "finalize" returning lsMsgBox
    exit method.
 end method "onSignalSearchFailed".
```

2. MainWindow Working-Storage 節を検索して、メッセージ ボックスに表示されるテキストの次の宣言を追加します。

```
   03 wsMsgSearchFailed    pic x(11)
                           value z"該当者なし".
```

9.10.5 "onSignalSaveFailed" の説明

このメソッドは、"onSignalSearchFailed" メソッドに似ていますが、SymbolTable からファイルの名前を取得して、

それを MessageBox に表示されるテキストの一部として使用します。

phonebook プロジェクトをリビルドして、再実行してください。これで、phonebook チュートリアルは終わりです。さらに詳しく知りたい場合は、MainApplication クラス (tpltapp.cbl) の "onSignal..." メソッドにブレークポイントを設定してみてください。すると、MainWindow が実行のどの時点でシグナルを送信するかを確認し、コードを実行して、各シグナルによって表わされる要求がどのように処理され、結果がどのように返されるかを確認することができます。

"onSignalSaveFailed" メソッドの追加例

1. 下のコードをコピーし、Phone アプリケーションの MainWindow クラス (tpltwin.cbl)の、END-OBJECT コード (プログラムの終わり近く) の直前に貼り付けます。

```
*>------------------------------------------------------
*> ウィンドウシグナルメソッド
*> このメソッドはウィンドウが Searchfailed シグナルを受け取ったときに
*> いつでも起動され、要素がファイルを保存しようとしたが失敗したことを表す
*>------------------------------------------------------
 method-id. "onSignalSaveFailed".

 local-storage section.
 01 lsString              object reference.
 01 lsMsgBox              object reference.
 01 lsLength              pic x(4) comp-5.
 01 lsSize                pic x(4) comp-5.
 01 lsBuffer              pic x(300).

 linkage section.
 01 lnkDataBlock           object reference.
```

```
 procedure division using lnkDataBlock.
*----ユーザコード．ファイル保存の失敗に対応するためにコードを追加する
*    デフォルト コードがメッセージ ボックスを表示する
    move spaces to lsBuffer
    move wsMsgSaveFailed to lsBuffer
    move length of wsMsgSaveFailed to lsLength
    add 2 to lsLength
    fetch value lsString from table lnkDataBlock
                      at symbol filename-key
    if lsString <> null
       compute lsSize = length of lsBuffer - lsLength
       invoke lsString "getValueWithSize"
              using lsSize
              returning lsBuffer(lsLength:)
       invoke lsString "size" returning lsSize
       add lsSize to lsLength
       if lsLength > length of lsBuffer
          move length of lsBuffer to lslength
    end-if
       move low-value to lsBuffer(lsLength:1)
       add 1 to lsLength
       move x"00" to lsBuffer(lsLength:1)
    end-if
    invoke MessageBox "new" using self returning lsMsgBox
    invoke lsMsgBox "setMessageZ" using lsBuffer
```

```
            invoke lsMsgBox "show"

            invoke lsMsgBox "finalize" returning lsMsgBox

        *----ユーザ終り.

            exit method.

         end method "onSignalSaveFailed".
```

2. MainWindow Working-Storage 節を検索して、メッセージ ボックスに表示されるテキストの次の宣言を追加します。

```
            03 wsMsgSaveFailed      pic x(19)

                                    value z"ファイルが保存できない".
```

9.11 まとめ

このチュートリアルでは、Windows リソースと GUI アプリケーション テンプレートを使用して、単純な GUI インタフェースを作成する方法を学習しました。

第10章 要件ベースのボキャブラリのチュートリアル

要件ベースのボキャブラリは、Micro Focus Object COBOL に固有の機能です。要件ベースのボキャブラリを使うと、対象となるアプリケーションの種類に関連した新しい文や関数を作成することができます。ここでは、継承のチュートリアルで紹介した銀行口座クラスを使って新しい機能のデモンストレーションを行います。 このチュートリアルは次のセッションからできています。

1. Account クラスのボキャブラリ
2. 要件ベースのボキャブラリの使い方
3. まとめ

学習時間：10分

10.1 Account クラスのボキャブラリ

ここでは、継承のチュートリアルで使用したものに似たアプリケーション例をロードします。唯一の違いは、ドライバ プログラム bank2.cbl にあります。これは、INVOKE 文ではなく、account クラスに定義されたボキャブラリを使用して、これらにメッセ ージを送信します。

Bank ボキャブラリの例の表示

1. NetExpress 統合開発環境を開始して、Vocabulary プロジェクト (bank¥vocab.app) をロードします。
2. プロジェクトをリビルドします。
3. bank2.cbl のビューを [Project:vocab.app] ウィンドウに拡張します。

   このプログラムは、2 つのコピーファイル account.if と saccount.if を使用します。これらは、Account および SavingsAccount の各クラスで使用する新しいボキャブラリを定義するクラス インタフェース ファイルです。Account のメソッドは、すべての型の account によって、インタフェースの変更なしに継承されるため、account.if で定義されたボキャブラリは、すべての account に適用されます。 saccount.if ファイルは、SavingsAccount (および SavingsAccount のサブクラスである HighRateAccount) に適切な追加ボキャブラリを定義します。

4. account.if がエディタ ウィンドウに表示されない場合は、名前を右ボタンでクリックして、ポップアップ メニューの [コピーファイルを表示] をクリックします。

class-id 見出しを確認します。これは、account.if を外部クラスとして定義します。コンパイラは、外部クラスのコードを何も生成しません。

5. "openAccount" メソッドまで下方向にスクロールします。

   account.cbl の "openAccount" と同じ method-id、linkage section、および procedure division 見出しがあります。ただし、linkage section には実行可能な文がありません。代わりに、新しい文 Create の構文定義があります。この定義は、invoked as という句から始まります。

これで、ボキャブラリのチュートリアルの最初のセッションは終わりです。次のセッションでは、accounts ボキャブラリを使用するいくつかのコードをアニメートします。

10.2 要件ベースのボキャブラリの使い方

このセッションでは、継承のチュートリアルでアニメートした bank1.cbl とまったく同じ機能を持つ bank2.cbl をアニメートします。ただし、このプログラムは、このチュートリアルの前のセッションで検討したインタフェース ファイルに定義されている新しい文を使用して、これらの機能を実行しています。

ボキャブラリ例のソース コードのアニメート

1. Animator を開始します。

   テキスト ウィンドウには、bank2.cbl のコードが、タグ B002 (**move "Dan" to aCustomer**) の下の文が実行に備えて強調表示された状態で示されます。

2. タグ B002 の下の行を実行します。

   新しい CheckAccount を開くメッセージが、account.if に定義されている新しい文 create によって送信されます。実行は、Account の "openAccount" メソッドに切り替わります (CheckAccount は、Account から "openAccount" メソッドを継承します)。

3. account 作成コードを、実行が bank2.int のタグ B003 の下の行に戻るまで実行します。

4. create 文の定義を確認できます。この定義は、account.if ファイルにあります。Ctrl + Home キーを押して Bank2 の一番上に進み、**copy "ACCOUNT.IF"** という行を探します。デフォルトでは、編集ウィンドウにはコピーファイル内のコードが表示されます。タグ 1001 の下の行 (**method-id. "openAccount"**) が表示されるまで Page-Down キーを押してください。

   procedure division 見出しの下には、invoked as 句があります。大かっこ [ ] 内の語はノイズ ワードで、読みやすくために入っています。create 文を使用するときに、[ ] 内の語を含める必要はありません。角かっこ <> 内の語は、<self> を除いて、ルーチンへのパラメータです。<self> は、メッセージを受け取るオブジェクトへの参照オブジェクトです。文の定義にあるパラメータはすべて、procedure division 見出しの using

および returning 句にあります。procedure division 見出しと linkage section は、Account クラス自体の openAccount メソッドにあるものとまったく同じです。

5. 現在の実行点にカーソルを返します。

6. タグ B008 に達するまで Bank2 のコードを実行します。

   コードを実行すると、SavingsAccounts および CheckAccounts に同じ文が使用される様子を見ることができます。

7. B008 の下の行 (**`if balance(asAccount)...`**) を実行します。

   この条件文は、balance 関数を使用します。これは、関数定義の例です。これは、固有 COBOL 関数と同様の方法で、たとえば条件文において、または compute 文の一部として使用できます。この関数は、このチュートリアルで使用した他の文とともに account.if に定義されています。

8. stop run に達するまで、コードを実行します。

これで、ボキャブラリのチュートリアルは終わりです。account.if と saccount.if を検討すると、作成するオブジェクトにボキャブラリを定義する方法を確認できます。

10.3 まとめ

ここでは、次のことを学習しました。

- 新しい文を定義してクライアントプログラムに対して有効にする方法
- クラスセットに定義された文の使用方法

第11章 プログラムでのオブジェクトの使い方

オブジェクトは、それ自身がオブジェクト指向ではないアプリケーションの中でも使用できます。作成したプログラムが、Object COBOL のクラス プログラムまたは Object COBOL オブジェクトを使う通常の COBOL プログラムのどちらであっても、オブジェクトの使用に関して同じ規則が適用されます。オブジェクトを使用するプログラムには、次のような共通する特徴があります。

- CLASS-CONTROL 段落を持っている
- OBJECT REFERENCE 型のデータ項目を 1 つまたは複数宣言している
- INVOKE 文を使ってメッセージを送る

一般にオブジェクトを使用するプログラムは、必要なインスタンス オブジェクトを作成するために、1 つまたは複数のクラス オブジェクトにメッセージを送ります。作成した各インスタンス オブジェクトについて、オブジェクト ハンドルを入れるためのオブジェクト参照を宣言しなければなりません。ただし、オブジェクト参照内のオブジェクトハンドルは、使い終わったら新しいものを上書きできます。使わなくなったオブジェクトは "finalize" メッセージを送って廃棄する必要があります。

オブジェクトを使うプログラムは、次の手順で作成します。

1. 使用するクラスを登録する
2. オブジェクト参照を宣言する
3. クラスおよびインスタンス オブジェクトにメッセージを送信する
4. 新しいオブジェクト インスタンスを作成する
5. オブジェクトが不要になったときに廃棄する

関連項目

- 2.1 オブジェクト
- 第 11 章: プログラムでのオブジェクトの使い方

11.1 クラスの登録

Object COBOL のクラスは、使用する前に登録する必要があります。これには、クラスを使用するすべてのプログラムの CLASS-CONTROL 段落で、クラスを宣言しなければなりません。これによって、クラスの名前がクラスプログラムの実行可能ファイルとリンクされます。プログラムがクラスである場合は、その CLASS-CONTROL 段落にもクラスとスーパークラスのエントリを含む必要があります。

クラスを登録すると、クラス オブジェクトとしてクラスを効果的に使用できるようになります。CLASS-CONTROL 段落のクラスの名前が、オブジェクト参照データ項目として自動的に宣言されます。これによって、クラス オブジェクトにメッセージを送ることができます。

クラスの登録方法

エントリを CLASS-CONTROL 段落に追加し、クラス名（ユーザのプログラム内部で使われているもの）を外部ファイル名（クラスのコードを含むもの）にリンクさせます。

例

この例では、クラス CharacterArray と myClass を登録する Class-Control 段落を示します。Class-Control 段落は新規オブジェクト節 (Object Section) の一部で、環境部 (Environment Division) の最後にあります。

```
class-control.

   CharacterArray is class "chararry"

   myClass is class "myclass"

   .
```

ファイル名は最後の二重引用符にあります。ファイル名は小文字で入力する必要があります。これは大文字・小文字のファイル名を区別するオペレーティング システムへの対応に備えて互換性を保つためです。

11.2 オブジェクト参照の宣言

OBJECT REFERENCE 型を使用するデータ項目を宣言して、使用する任意のインスタンス オブジェクトにハンドルを保存する必要があります。オブジェクト参照の宣言は、Working-Storage 節、Local-Storage 節、Linkage 節および、プログラムの Object-Storage 節で、行えます。クラス オブジェクトのオブジェクト参照は、クラスを登録する際に自動的に宣言されます。

オブジェクト参照は、ポインタを使用したデータ項目に似ています。これらの表示は、プログラマではなくコンパイラによって決まります。また、オブジェクト参照に対して、COBOL データ操作のセットすべてを実行することはできません。 次のことが許可されます。

- オブジェクト参照を INVOKE 文のターゲットとして使用する
- オブジェクト参照をパラメータとしてプログラムまたはメソッドへ渡す
- あるオブジェクト参照を、別のオブジェクト参照値またはヌルに設定する
- あるオブジェクト参照と、別のオブジェクト参照値またはヌルとを、対等に比較する

例

この例では、オブジェクト参照を保持するデータ項目の宣言方法を示します。

```
anObject          object reference.
```

宣言されたデータ項目 (anObject) は 1 つのオブジェクト処理をいずれの種類のクラスやインスタンス オブジェクトに保持することができます。

関連項目

- 2.1 オブジェクト
- 第 11 章: プログラムでのオブジェクトの使い方
- 11.1 クラスの登録

11.3 メッセージの送信

オブジェクトのメソッドを実行するには、そのオブジェクトにメッセージを送ります。メッセージの送信は、オブジェクトを取得してアプリケーションで作業する方法です。 実行されたメソッドは、メッセージに対応した識別子を持ちます。INVOKE 文を使って、メッセージを送信します。

オブジェクトが理解できないメッセージは、認識および実行されるまで、メソッド継承チェーンに渡されます。メソッド継承チェーンについてオブジェクトのユーザは考慮する必要はありません。オブジェクトの実現に関する詳細に含まれます。

例

メッセージを送信するには、メッセージの受信者とメッセージ セレクタを指定します。受信者とはメッセージを送信するオブジェクトで、常に OBJECT REFERENCE 型 のデータ項目で指定されます。たとえば、メッセージ "sayHi" を送信するには、次を記述します。

```
invoke anObject "sayHi"
```

次の例では、使用するパラメータと戻されたパラメータの両方を持つ送信メッセージを示します。

```
invoke anObject "fetchData" using aKey returning anObject
```

2 つ以上の USING パラメータを指定できますが、指定できる RETURNING パラメータは 1 つだけです。

次のコード例では、メッセージ名が実行時に動的に選択されたメッセージの送信方法を示します。PIC X(*n*) データ項目が使用でき、ここでは *n* の数値が、格納するメッセージ名に 1 スペースを加えた数値よりも大きくなります。

この例では、aMessage を PIC X(32) として宣言します。

```
move "new " to aMessage

...

invoke myClass aMessage returning myObject
```

パラメータを送るのと結果を受け取るのに同じ変数を使ってメッセージを送信した場合は、結果が予測できません。たとえば、次のようなコーディングをしないでください。

```
invoke myObject "add" using aValue returning aValue
```

関連項目

- 2.1 オブジェクト
- 2.3 メソッド
- 2.6 メッセージと結合
- 第 11 章: プログラムでのオブジェクトの使い方
- 11.3 メッセージの送信
- 12.4.4 クラス メソッド
- 12.5.4 インスタンス メソッド
- 12.6.1 メソッド プログラム
- 12.6.3 自分宛てのメッセージ送信

11.4 新しいオブジェクト インスタンスの作成

新しいオブジェクトを作成すると、OO RTS は OBJECT-STORAGE 節で宣言された変数のために、オブジェクト ハンドルと記憶領域を割り当てます。クラスでは、新しいオブジェクト インスタンスを作成するための独自のメソッドを用意するか、親クラスからメソッドを継承します。

新しいインスタンス オブジェクトの作成方法

1. OBJECT REFERENCE 型のデータ項目を宣言し、オブジェクト ハンドルを作成時にオブジェクトに保持します。
2. クラスの新規インスタンスを要求しているクラス オブジェクトに、メッセージを送信します。 送信されたメッセージは、クラスのパブリック インタフェースの一部です。 初期化パラメータを要求しないイン

スタンス オブジェクトでは、これは常に「新規」です。

例

この例では、オブジェクト参照を宣言し、クラス aClass の新規インスタンスにオブジェクト処理を格納するようオブジェクト参照を使用します。

```
class-control.
    aClass is class "classprg"
    ...
working-storage section.
   01  anObject          object reference.
   ...
procedure division.
   ...
   invoke aClass "new" returning anObject
```

この例では、aClass がオブジェクト参照 anObject にあるそれ自体の新規インスタンスに処理を戻します。

一部のクラスでは、作成時にインスタンスを初期化するため 2 つ以上のパラメータを必要とする場合があります。たとえば、OrderedCollection (クラス ライブラリで提供される一部) のインスタンスを作成するには、作成されるオブジェクトの初期サイズを提供する必要があります。次のコード例では、OrderedCollection の新規インスタンスを作成します。

```
invoke OrderedCollection "ofReferences"
                        using aSize
                    returning anOrderedCollection
```

OrderedCollection を作成するメソッドの 1 つが "ofReferences" です。新規インスタンスを作成するためのメソッドとそれらの使用方法を検索するには、常にクラスへのパブリック インタフェースを確認する必要があります。

11.5 オブジェクトの廃棄

アプリケーションがオブジェクトを使い終わった時は、オブジェクトを廃棄し、オブジェクトが使用していたメモリを解放する必要があります。Object COBOL では、自動廃棄ガーベージ コレクション (automatic garbage collection) が実現されていないからです。オブジェクトを廃棄する時期の決定およびオブジェクトの廃棄は、次の指針に従っ

て行います。

- オブジェクトを作成したものが廃棄の責任を持つ

  これは、現在の実行単位が終了した後もオブジェクトが存在する、永続性を利用したアプリケーションで特に重要です。しかし、アプリケーションの他の部分にオブジェクトを渡した場合、オブジェクトが引き続き使用中かどうかがわからない場合があります。他のオブジェクトに渡されたオブジェクトが廃棄できるかどうかを示すための規則を確立する必要があります。 例えば、Window のタイトルを設定するために Window に CharacterArray オブジェクトを渡す場合、CharacterArray は後で廃棄してかまいません。オブジェクトを任意のコレクションに格納する場合、それが入ったコレクションを廃棄せずに、オブジェクトだけを廃棄することはできません。 アプリケーションが使い終わってもまだ廃棄されていないオブジェクトは、メモリリーク構成しているといいます。

- すべてのオブジェクトは動的である

  Object COBOL の今回のバージョンには、C++ の静的オブジェクトに相当するもの (1つのメソッドの範囲内で作成され、終了時に自動的に廃棄される) はありません。一度作成したオブジェクトは、そのオブジェクトハンドルが入ったデータ項目が廃棄されたり、範囲外になっても、明示的に廃棄されるまで存在し続けます。

- 他のオブジェクトが参照しているオブジェクトは廃棄しない

  オブジェクトを廃棄したら、そのオブジェクトを引き続き指している参照にはメッセージを送らないようにします。メッセージを送った場合、予想できない結果が起こります。

例えば、次のようなコーディングはしないでください。

```
set objectA to objectB

invoke objectB "finalize" returning objectB

invoke objectA "aMessage"
```

objectAに入っているオブジェクトハンドルは、存在していないオブジェクトを指しています。次の例には、上の例と同じエラーがありますが、気がつきにくいコーディングです。

```
invoke objectA "setValue" using objectB

invoke objectB "finalize" returning objectB
```

objectA が objectB のコピーを自身のために作成し、objectB へのハンドルを持っていないかぎり、objectA は無効なオブジェクト ハンドルを持つことになります。

付属のクラス ライブラリにあるすべてのオブジェクトは、コレクションに格納された要素を除き、パラメータとして渡されたすべてのオブジェクトのコピーを作成します。ただし、コレクションに格納された要素はこの限りでは

ありません。結果として、クラス ライブラリ オブジェクトにパラメータとして渡したオブジェクトを廃棄することができます。

参照先オブジェクトが最終化された後に、オブジェクト ハンドルにメッセージを送信すると、予期しない結果が起こります。

- オブジェクト ハンドルが Object COBOL RTS に再使用されていない場合は、アプリケーションはランタイム エラーによって停止します。
- オブジェクト ハンドルが Object COBOL RTS に再使用されている場合は、メッセージは間違ったオブジェクトに送信されます。 オブジェクトがメッセージを認識する場合は、何らかの誤りがある旨を知らせるようなエラーはありません。 最終化されたオブジェクト参照の再使用を防止する、Object COBOL ランタイム切り替えがあります。これによって、最終化されたオブジェクトにメッセージを送信しようとすると、必ずランタイム エラーが発生します。

関連項目

- 2.1 オブジェクト
- 第 11 章: プログラムでのオブジェクトの使い方
- 22.2.4 メモリリークの検索

11.5.1 コレクション オブジェクトとその要素を廃棄する方法

"deepFinalize" メッセージに送信します。

例

この例では、"deepFinalize" メッセージを使用して集合と集合を含むすべての要素を廃棄する方法を示します。

```
invoke aCollection "deepFinalize" returning aCollection
```

"deepFinalize" メソッドが集合にある要素すべてを廃棄し、次にコレクションそのものを廃棄します。

関連項目

- 11.5 オブジェクトの廃棄

11.5.2 シングル オブジェクトを廃棄する方法

"finalize" メッセージに送信します。

例

この例では、"finalize" メッセージを使用してオブジェクトを廃棄する方法を示します。

```
invoke anObject "finalize" returning anObject
```

これは Base で実行されるデフォルト メソッドです。ヌル オブジェクト処理を戻します。

関連項目

- 11.5 オブジェクトの廃棄

第12章 クラス プログラム

OO アプリケーションを作成するには、Object COBOL のクラス ライブラリに入っているクラスに加え、自分でクラスを作成する必要があります。この章では、クラス プログラムの作成に関する規則を短いコード例を示しながら説明します。

12.1 概要

Micro Focus またはその他のクラス ライブラリで指定した以外のオブジェクトを作成するために、Object COBOL クラスを作成することが必要です。クラスを作成するときは、次の 3 つの明確なエンティティを理解することが必要です。

- クラス
- クラス オブジェクト
- インスタンス オブジェクト

クラスとクラス オブジェクトを混同しないように注意が必要です。互いに関連がありますが、同じものではありません。クラスはソース コード ファイルと実行可能コードファイルとして存在しますが、クラス オブジェクトは実行時にクラスがロードされたときにだけ存在します。

オブジェクトの設計は、本書の対象外です。オブジェクトに関連する文献をお読みになって、OO の分析を行い、可能な場合は設計のトレーニングを行うことをお勧めします。

クラスを作成する前に、別のクラス (スーパークラス) から機能を継承すると便利かどうかを、識別する必要があります。たとえば、アプリケーションのメイン ウィンドウであるオブジェクトを作成している場合、おそらくクラス ライブラリの Window クラスから継承します。継承する特殊な機能をもつクラスがない場合でも、通常は自分のクラスを基本となるサブクラスにし、新しいルート クラスの作成が複雑にならないようにします。

クラスの構造についての詳細は、12.2 クラスの構造を参照してください。次のトピックがあります。

- 12.2.1 クラス プログラムの構造
- 12.2.2 クラスの名前指定と設定の継承
- 12.2.3 共有記憶の宣言
- 12.2.4 クラス拡張
- 12.2.5 オブジェクト指向とマルチスレッド

継承についての詳細は、12.3 継承を参照してください。次のトピックがあります。

- 12.3.1 メソッドの継承方法
- 12.3.2 データの継承方法
- 12.3.3 ソース コードのデータ継承
- 12.3.4 実行時のデータ継承

クラス オブジェクト動作のプログラミングについての詳細は、12.4 クラス オブジェクト動作のプログラミングを参照してください。次のトピックがあります。

- 12.4.1 クラス オブジェクト プログラム
- 12.4.2 クラス データ
- 12.4.3 クラスの初期化
- 12.4.4 クラス メソッド

インスタンス オブジェクト動作のプログラミングについての詳細は、12.5 インスタンス オブジェクト動作のプログラミングを参照してください。次のトピックがあります。

- 12.5.1 インスタンス オブジェクト
- 12.5.2 インスタンス データ
- 12.5.3 インスタンスの初期化
- 12.5.4 インスタンス メソッド

プログラミング メソッドについての詳細は、12.6 プログラミング メソッドを参照してください。次のトピックがあります。

- 12.6.1 メソッド プログラム
- 12.6.2 メソッド データとパラメータ
- 12.6.3 自分宛てのメッセージ送信
- 12.6.4 新規インスタンスの作成方法

関連項目

- 2.2 クラス
- 第 11 章: プログラムでのオブジェクトの使い方

12.2 クラスの構造

ここでは、クラスの構造について説明します。

12.2.1 クラス プログラムの構造

各クラスは、2 種類のオブジェクトの動作を説明します。

- クラス オブジェクト
- インスタンス オブジェクト

アプリケーションには、各クラス オブジェクトのオカレンスは1つまでです。しかし、そのインスタンスのオカレンスは複数存在できます。クラス オブジェクトのおもな機能は、インスタンスを作成することですが、他の動作をすることもあります。
クラス オブジェクトは入れ子プログラムのセcットから構成されます (図12-1を参照)。

図 12-1 クラス プログラムの構造

Shared data & definitions
(共有データと定義)

共有データと定義は、クラスプログラムのすべての部分にアクセスすることができます。データはクラスプログラム データ部で宣言されます。定義には CLASS-ID 見出しのデータ継承のための設定、環境部の設定が含まれ、CLASS-CONTROL 段落で登録されたすべてのクラスが入れられます。

Class initialization code
(クラス初期化コード)

クラス オブジェクトの中に任意に「initializeClass」メソッドを含むことができます。これはクラスが初めて読み込まれたときに実行され (アプリケーションが実行されるたびに一度だけ)、任意の初期設定コードのために使用することができます。

Class Object
(クラス オブジェクト)

クラス オブジェクトは CLASS-OBJECT 見出しで始まり、END CLASS-OBJECT 見出しで終わる入れ子のプログラムです。クラス データを宣言するための Object-storage 節と 1 つまたは複数のクラス メソッドを任意に含むことができます。クラス プログラム内にクラス オブジェクトを含む必要はありません。一部のクラスはインスタンス動作のみを定義します。

Object (オブジェクト)

インスタンス オブジェクトは OBJECT 見出しで始まり、END OBJECT 見出しで終わる入れ子のプログラムです。インスタンス データを宣言するための Object-storage 節と 1 つまたは複数のインスタンス メソッドを任意に含むことができます。クラス プログラム内にインスタンス オブジェクトを含む必要はありません。一部のクラスはクラス動作のみを定義します。

クラス アウトラインの例

次の例は、Exampleクラスのアウトラインで、クラスの主な特徴を示しています。省略記号 (...) はコードが省略されている部分、インデントは入れ子のレベルを意味します。

```
class-id. Example inherits      *> クラスの識別。
          from Base.            *> データの継承も設定する(この例にはない)
                                *>
                                *>
 environment division.          *> 環境部の見出しは任意。
                                *>
                                *> MFまたはANSI85のCOBOLで適用できる環境
                                *> 部のすべての節が可能。
```

```
                                *> ここでは示さない
                                *>
                                *>
...
object section.                 *> OO仕様の環境情報を含んでいる節。
                                *>
                                *>
class-control.                  *> class-control段落は、各クラスの
                                *> 実行形式を含んでいるファイルに
                                *> 対して名前を与える
                                *>
                                *>
Example is class "exmp"         > Examp クラス用の実行形式は
                                *> ファイルexmpにある。
                                *>
Base is class "base"
CharacterArray is class "chararry"
.                               *> ピリオドで段落を終了。
data division.                  *> データ部の見出しは任意。
                                *>
                                *> MFまたはANSI85のCOBOLで適用できる
                                *> すべての節が可能。
                                *> ここでは示さない。
                                *>
```

```
...
working-storage section.
...
    class-object                *> クラス オブジェクトのはじめ。
                                *>
    object-storage section.     *> クラス オブジェクトデータの定義
    ...


    ...


        method-id. "new".       *> クラスメソッド "new"のはじめ。
        ...
        end method "new".       *> クラスメソッド "new"の終わり。
   end class-object.            *> クラス オブジェクトの終わり。


   object.                      *> クラスのインスタンスの操作を
                                *> 定義するコードのはじめ。
                                *>
                                *>
   object-storage section.      *> インスタンスデータの定義
    ...
        method-id. "sayHello".  *> インスタンスメソッド
                                *> "sayHello"のはじめ。
        ...
```

```
        end method "sayHello".   *> インスタンスメソッドの終わり
    end object.                  *> インスタンス オブジェクト用の
                                 *> コードの終わり。
end class Example.
```

12.2.2 クラスの名前指定と設定の継承

クラスのソース コードは CLASS-ID 見出しで始まり、END CLASS 見出しで終わります。CLASS-ID はクラスの名前を指定します。スーパークラスがある場合は、スーパークラスの名前も指定します。ルート クラスにはスーパークラスがありません。CLASS-ID 見出しは、このクラスがデータを継承するかどうか、自身のデータがプライベートであるか、保護されているかも決めます。

CLASS-ID 見出しで指定されたクラス自身およびスーパークラスの両方について、Class-control 段落の中にエントリを含む必要があります。

方法および例については、次を参照してください。

- 12.3.3.1 データを継承しないクラスの宣言方法
- 12.3.3.2 データを継承するクラスの宣言方法
- 12.3.3.3 プライベート データをもつクラスの宣言方法
- 12.3.3.4 保護データをもつクラスの宣言方法
- 12.3.3.5 ルート クラスの宣言方法

関連項目

- 2.4 継承
- 11.1 クラスの登録
- 12.3.1 メソッドの継承方法
- 12.3.2 データの継承方法

12.2.3 共有記憶の宣言

クラス オブジェクトの始まりの前にクラス データ部で宣言されたデータはすべて、クラス内で入れ子されたすべてのプログラムの適用範囲となります。つまり、ここで宣言されたデータは、クラス メソッドおよびインスタンス メソッドで見ることができます。結果として、クラスのこの部分の Working-Storage 節を、インスタンスを初期化する

共通データとして、またはインスタンスすべてにグローバルなデータのキャッシュとして使用することができます。

注記 :

このような Working-Storage の使用方法は Micro Focus の独自機能です。現在、OO COBOL に対して提案された ANSI 標準の一部ではありません。

共有記憶の例

この例は、クラスの共有記憶を示します。

```
class-id. Example inherits
          from Base.
* 環境部と Class-control 節
 ...
 data division.                *> データ部の見出しは任意。
                              *>
 file section.                 *> File 節と Working-storage 節で宣言されたデータは共有される。
 ...                           *>
 working-storage section.      *>
 ...
```

- クラス オブジェクト コード

```
...
* インスタンス オブジェクト コード
...
 end class Example.
```

関連項目

- 12.4.2 クラス データ

- 12.4.3 クラスの初期化
- 12.5.2 インスタンス データ

12.2.4 クラス拡張

クラスの拡張では、元のソース コードを変更せずに Object COBOL に新しいメソッドを追加できます。クラス拡張とサブクラス作成の違いは、拡張はすべての既存のサブクラスによって継承される点にあります。

たとえば、クラス A にサブクラス クラス B がある場合、そのサブクラスを作成してクラス A に機能を追加することで、サブクラス C を作成することができます。しかし、クラス B はクラス A のサブクラスであるため、クラス C の機能を継承しません。クラス A をクラス拡張 X で拡張する場合、実行時の効果はクラス A を変更し、再コンパイルした場合と同じになります。クラス B はクラス X のすべての追加機能を継承します。

Class-Control 段落は、拡張クラスとそれが拡張するクラスへの参照を含む必要があります。WITH DATA 句を使用して、拡張クラスが元のクラスのクラスとインスタンスデータを直接、参照できるようにします。拡張クラスは、新規クラスまたは自身のインスタンス データを宣言できません。ただし、自身の Working-Storage を宣言することはできます。

クラス拡張を使用するには、クラス拡張が使用される前にアプリケーションを実行する時点で、COBOL CALL を挿入する必要があります。これによって、Object COBOL RTS に新規メソッドを登録します。

クラス拡張の例

```
class-id. ExtensionClass extend OriginalClass [with data].


 class-control.
    ExtensionClass is class "extend"
    OriginalClass is class "original"
```

- **追加のクラス制御エントリが必要。**

```
    ...
class-object.
```

- `OriginalClass` **に拡張するための新規クラス メソッド**

```
...
 end class-object.


 object.
* OriginalClass に拡張するための新規インスタンス メソッド
 ...
 end object.
 end class Example.
```

関連項目

- 11.1 クラスの登録

12.2.5 オブジェクト指向とマルチスレッド

複数のスレッドから起動できる任意の Object COBOL クラスは、REENTRANT コンパイラ指令でコンパイルする必要があります。REENTRANT コンパイラ指令は、クラスが複数のスレッドから起動するときにクラス初期化プロセスが正しく取り扱われるようにします。

この指令なしでコンパイルされるクラスは、同時に複数のスレッドから起動するときに、予測不可能で一貫しないものになってしまう可能性があります。

12.3 継承

ここでは、継承について説明します。ほとんどの Object COBOLプログラムはスーパークラスから継承しています。付属のクラス ライブラリにおける唯一の例外は Base クラスです。Base クラスは、ライブラリ内の他のすべてのクラスの継承ツリーのルートになります。自分で Object COBOL を作成する場合、ほとんどの場合は Base クラスまたは他のクラスのサブクラスとして作成します。

Base クラスまたはそのサブクラスのサブクラスではないクラスを作成する場合、Base によって提供される機能は継承しません。新しいインスタンスを作成するためのメソッド "new" は Behavior クラスの一部で、常に利用できますが、不必要なインスタンスを廃棄するためのメソッド (Base の "finalize") を自分で実現する必要があります。

クラス プログラムは次の 2 つをスーパークラスから継承できます。

- メソッド
- データ

次の各項では、これらについて説明します。

12.3.1 メソッドの継承方法

クラス オブジェクトは、スーパークラスのすべてのクラス メソッドを継承します。インスタンスは、スーパークラスのすべてのインスタンス メソッドを継承します。スーパークラスは、そのスーパークラスのすべてのクラスおよびインスタンス メソッドを継承します。したがって、オブジェクトは、ルートクラス (通常はBase) までのすべてのスーパークラスによって継承される、すべてのメソッドを継承することになります。

オブジェクトにメッセージを送信するときに、Object COBOL RTS はまず対応するメソッドを探すためにオブジェクトのクラスを見つけてから、スーパークラスを見つけます。検索パスはインスタンス オブジェクトとクラス オブジェクトとでは、若干、異なります。すべてのクラス オブジェクトがメタクラスのインスタンスであるからです。

メソッドの継承の詳細は以下のトピックを参照してください。

- 12.3.1.1 インスタンス オブジェクトのメソッドの継承
- 12.3.1.2 クラス オブジェクトのメソッドの継承

オブジェクトまたはそのスーパークラスによって実現されていないメソッドを起動しようとした場合、オブジェクトは "doesNotUnderstand" メッセージを送信します。これはデフォルトで例外を発生させます。 "doesNotUnderstand" の働きについては、12.3.1.3 理解できないメッセージを参照してください。

関連項目

- 2.3 メソッド
- 12.3.2 データの継承方法
- 12.4.4 クラス メソッド
- 12.5.4 インスタンス メソッド
- 12.6.1 メソッド プログラム

12.3.1.1 インスタンス オブジェクトのメソッドの継承

インスタンス オブジェクトへのメッセージを送信するときに、Object COBOL RTS は、まず、オブジェクトのクラス内で一致するメソッドを探そうとします。見つからない場合、メソッドが見つかるまでオブジェクトのスーパークラスを検索します。図 12-2 に、メッセージがとるパスを示します。

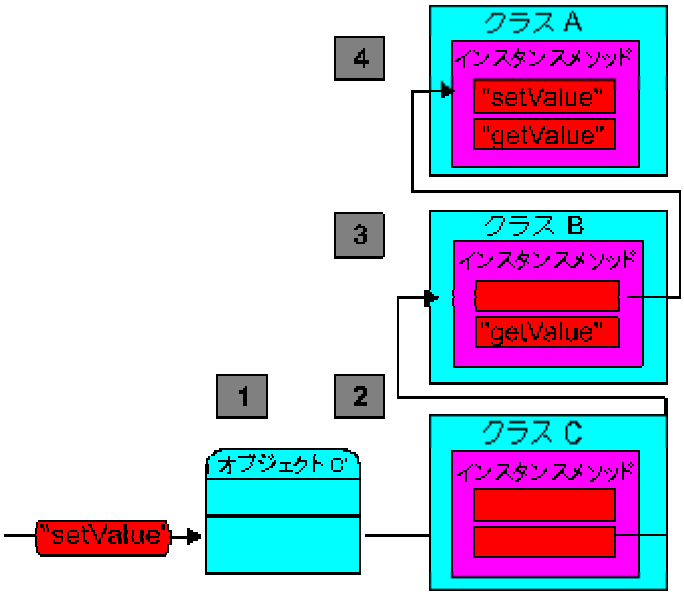


図 12-2 インスタンス オブジェクトのメソッドの継承

1. オブジェクト C' は、"setValue" メッセージを受け取ります。オブジェクト C' はクラス C のインスタンスです。

2. Object COBOL RTS は、クラス C の "setValue" インスタンス メソッドを探します。見つからない場合には、クラス B を探します。

3. クラス B は、いずれも "setValue" インスタンス メソッドを含みません。そのため、Object COBOL RTS はクラス A へメソッドを渡します。

4. クラス A は "setValue" メソッドを実行します。クラス A とクラス B は両方とも "getValue" メソッドを実現します。クラス C が代わりに "getValue" メッセージを送信した場合、クラス B 内で実現されたバージョンが実行されているはずです。サブクラスの動作を変更したいときは、すでにスーパークラスにすでにあるメソッドをいつでも実現し直すことができます。

12.3.1.2 クラス オブジェクトのメソッドの継承

クラス オブジェクトへのメッセージを送信するときに、Object COBOL RTS は最初にそのオブジェクトのクラスで一致するメソッドを探そうとします。探せなかった場合、メソッドが見つかるまでオブジェクトのスーパークラスを検索します。Object COBOL RTS がメソッドを見つけられずにルート クラスに達した場合、メタクラスのインスタンス メソッドを検索します。

メタクラス Behavior は Base のサブクラスであるため、メソッドが Behavior の中にない場合、Object COBOL RTS は Base のインスタンス メソッドを検索します。

図 12-3 に、Object COBOL RTS がクラス メソッドを検索するパスを示します。

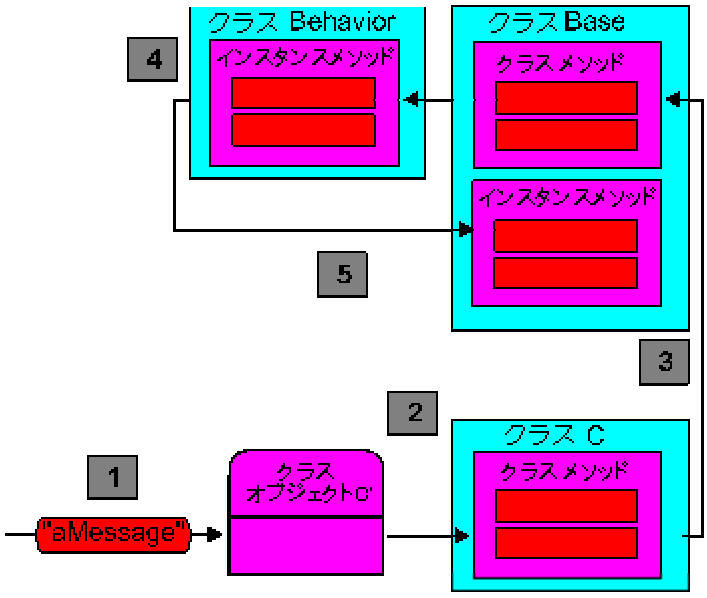


図 12-3 クラス オブジェクトのメソッドの継承

1. クラス オブジェクト C はメッセージを送信されます。
2. Object COBOL RTS は、クラス C のメッセージを検索します。
3. メソッドはクラス C で実現されません。そこで、Object COBOL RTS はルート (この場合は Base) に達するまでクラス C のスーパークラスを検索します。
4. メソッドは、どの Base でも実現されません。しかし、クラスはすべてメタクラス Behavior のインスタンスであるため、Object COBOL RTS は次に Behavior のインスタンス メソッドを検索します。この点が、クラス オブジェクトの検索手順がメソッドの検索手順と異なります。
5. Behavior は Base のサブクラスであるため、メソッドが Behavior のインスタンスを見つけることができない場合、検索は Base のインスタンス メソッドで完了します。

12.3.1.3 理解できないメッセージ

"Object COBOL RTS がオブジェクトまたはスーパークラスのクラスでメソッドを見つけられない" というメッセージをオブジェクトに送信すると、メッセージは 2 番目のメッセージ "doesNotUnderstand" をオブジェクトに送信します。オブジェクトは自分自身のバージョンの "doesNotUnderstand" を実現することができます。これによって、オブジェクトが理解できないメッセージを受け取ったときの特別な動作を指定できます。

代わりに、"doesNotUnderstand" を実現できない場合、Object COBOL RTS はオブジェクトのスーパークラスまでこのメッセージを検索します。Base は例外を発生させるデフォルトの "doesNotUnderstand" を実現します。図 12-4 を参照してください。

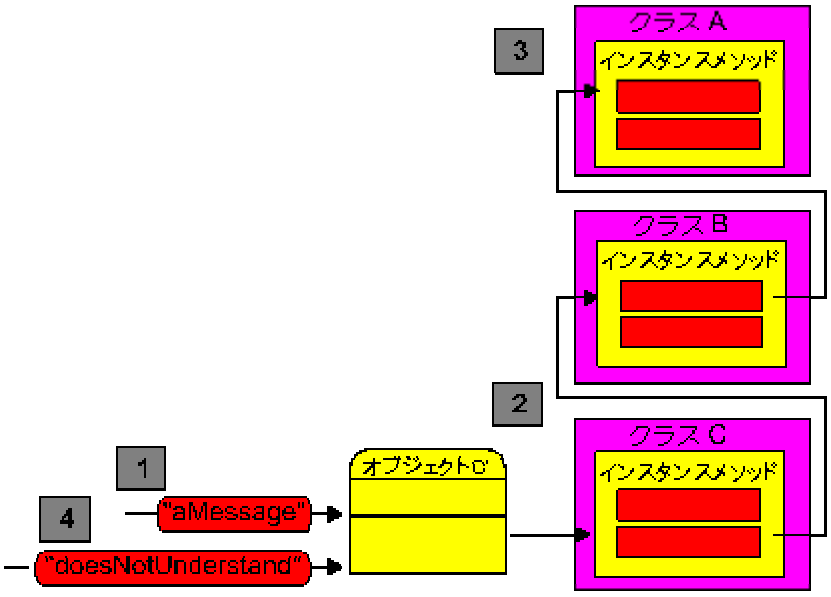


図 12-4 理解できないメッセージ

1. オブジェクト C'、クラス C のインスタンスはメッセージを送信されます。
2. メッセージはクラス C によって実現されません。そこで、Object COBOL RTS は、ルート クラス (通常は Base) に達するまで一致するメソッドのスーパークラスを検索します。
3. Object COBOL RTS は、一致するメソッドを検索できませんでした。
4. Object COBOL RTS は、メッセージ "doesNotUnderstand" を オブジェクト C' に送信します。"doesNotUnderstand" の実現を見つけるまで、クラス C とそのスーパークラスを検索します。

12.3.2 データの継承方法

データ継承は、スーパークラスの Object-Storage 節で宣言されたデータ項目に、クラスがアクセスできるようにします。サブクラスがデータを継承するとき、サブクラスのメソッドは継承されたデータを直接、実際に操作することができます。サブクラスがデータを継承しないとき、記憶域はスーパークラスで宣言されたデータに対して割り当てられたままですが、サブクラス メソッドは直接、記憶域を操作できなくなります。

データ継承を制御する方法と実行時の結果について、次のトピックで説明します。

- 12.3.3 ソース コードのデータ継承
- 12.3.4 実行時のデータ継承

関連項目

- 2.4 継承

- 12.3.1 メソッドの継承方法
- 12.3.2 データの継承方法
- 12.4.2 クラス データ
- 12.5.2 インスタンス データ

12.3.3 ソース コードのデータ継承

クラスを作成するとき、データは次のいずれかになります。

- プライベート

  クラス オブジェクトのみがクラス データを表示できます。また、そのクラスのインスタンス オブジェクトのみがインスタンス データを表示できます。

- 保護 (またはいわゆる制限付き)

  サブクラスは、このクラスの Object-Storage 節で宣言されたデータを任意に継承することができます。

保護データでクラスをコンパイルするとき、COBOL Checker は 2 つのコピーファイルを生成します。*filename*.cls および *filename*.ins です (ただし、*filename* はソース コード ファイルです)。コピーファイルは、クラスが継承したデータのすべての定義で始まり、クラスで宣言されたすべてのクラスとインスタンス データの定義が続きます。

データ継承を実現するためのコピーファイルの使用には、次の暗黙的方法があります。

- クラスにデータを継承させた場合、スーパークラスから順にコンパイルする必要があります。したがって、サブクラスをコンパイルするときに.ins および .cls ファイルを使用できます。
- スーパークラス用の.cls および.ins ファイルは、データを継承するクラスと同じディレクトリにあるか、*COBCPY*環境変数で指定されたディレクトリにある必要があります。

データの継承を制御するには、次のような方法があります。

- 12.3.3.1 データを継承しないクラスの宣言方法
- 12.3.3.2 データを継承するクラスの宣言方法
- 12.3.3.3 プライベート データをもつクラスの宣言方法
- 12.3.3.4 保護データをもつクラスの宣言方法
- 12.3.3.5 ルート クラスの宣言方法

それぞれの方法について説明します。

12.3.3.1 データを継承しないクラスの宣言方法

データを継承しないクラスを宣言するには、CLASS-ID 見出しから、WITH DATA を除外します。ここでは、クラス B がクラス A で宣言されたすべてのデータを継承しないようにする CLASS-ID 文を使ったクラスの例を示します。

```
class-id. B data is protected

        inherits from A.
```

データを継承しないクラスは、スーパークラスにあるメソッドを通じてデータにアクセスすることだけができます。メッセージを自身に送信することによって、スーパークラスにあるメソッドを起動します。次にメソッドの継承機構がスーパークラスにあるメソッドを探します。

12.3.3.2 データを継承するクラスの宣言方法

データを継承するクラスを宣言するには、CLASS-ID 見出しで、WITH DATA と指定します。ここでは、クラス B がクラス A で宣言されたデータすべてを継承するようにする CLASS-ID 文を使ったクラスの例を示します。

```
class-id. B data is protected

        inherits from A

        with data.
```

- **クラス用にコード化します。**

```
...

 end class B.
```

注記：

クラス A は保護データを使って宣言する必要があります。これを行わない場合は、クラス B をコンパイルしようとするとエラーが発生します。

12.3.3.3 プライベート データをもつクラスの宣言方法

プライベート データを持つクラスを宣言するには 、CLASS-ID 見出しで、DATA IS PRIVATE と指定します。ここでは、プライベート データを持つクラスを指定する CLASS-ID 文を使ったクラスの例を示します。

```
class-id. B data is private

         inherits from A.
```

- **クラス用にコード化します。**

```
...

 end class B.
```

ここでは B のサブクラスがそのデータを継承しないようにします。代わりに、PRIVATE をデフォルトとするように、データ節を合わせて省略できます。たとえば次のとおりです。

```
class-id. B inherits from A.

* Code for the class

 ...

 end class B.
```

12.3.3.4 保護データをもつクラスの宣言方法

クラスが保護データ (いわゆる制限付きデータ) と宣言されている場合は、サブクラスはオプションでクラスからデータを継承できます。保護データを持つクラスを宣言するには、CLASS-ID 見出しで、DATA IS PROTECTED と指定します。PROTECTED の同義語として、RESTRICTED を使うことができます。ここでは、クラスで宣言されたデータをサブクラスで継承できるようにする CLASS-ID 文を使ったクラスの例を、示します。

```
class-id. B data is protected

        inherits from A.
```

- **クラス用にコード化します。**

```
...

 end class B.
```

これによって、クラス B で宣言されたデータが、WITH DATA を継承する B のどのサブクラスへも直接アクセスできるようになります。

12.3.3.5 ルート クラスの宣言方法

ルート クラスを宣言するには、CLASS-ID 見出しから、INHERITS FROM 句を除外します。ここでは、ルート クラス (継承なし) としてクラスを宣言する CLASS-ID 文を使ったクラスの例を示します。

```
class-id. B data is protected.
```

Base からメソッドを継承できないため、ルート クラスはインスタンスを最終化するために独自のメソッドを使用する必要があります。これはメタクラスで提供されたように、新しいインスタンスを作成するために "new" を継承します。

関連項目

- 12.2.2 クラスの名前指定と設定の継承
- 12.3.1 メソッドの継承方法
- 12.3.2 データの継承方法

12.3.4 実行時のデータ継承

スーパークラスからデータを継承するとき、コンパイラはスーパークラスからすべてのクラス オブジェクト データ定義のコピーをクラスの Object-Storage 節に入れます。また、すべてのインスタンス オブジェクト データ定義のコピーをインスタンスの Object-Storage 節に入れます。

図 12-5 は 3 つのクラス A、B、C を示しています。C は B からデータなしで継承し、B は A からデータ付きで継承します。クラス A はインスタンスメソッド "setInDataA" を実現し、instanceItemA というデータ項目に値を設定します。点線の右側には、A1、B1、C1 の 3 つのインスタンス オブジェクト があります。これらはクラス オブジェクト A、B、C によって実行時に作成されます。クラス B はデータを継承しますが、クラス C はデータを継承しません。グレーの領域は、スーパークラスで実現されるメソッドを通してのみアクセスできるスーパークラス データに対して、割り当てられます。

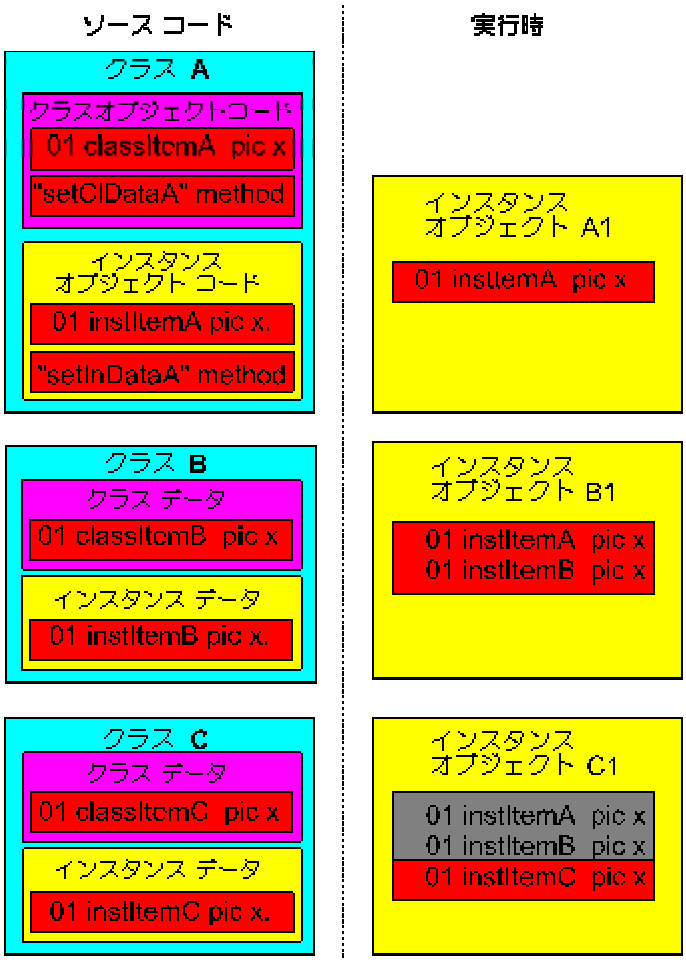


図 12-5 実行時のデータ継承

オブジェクト B1 がメッセージ "setInDataA" を受け取った場合、メッセージは A で実現されたインスタンス メソッドを起動します。B はこのメソッドを実現していないからです。しかし、更新されたデータ項目はインスタンス オブジェクトの B1 の記憶域に保持された instanceItemA です。

クラス B は A からデータ付きで継承するので、クラス B にあるソース コードも instItemA を直接、操作できます。たとえば、クラス B のプログラマは次のような文が入ったインスタンスをコーディングできます。

```
move "!" to instItemA
```

Object C1 にも、実行時に Object-Storage 節で instItemA および instItemB のためにメモリが割当てられます。しかし、データ付きで継承していないので、クラス C をコンパイルするときに、COBOL Checker はこれらの項目の定義を知ることができません。クラス オブジェクト C に "setInDataA" メッセージを送ることによってのみ、instItemA の値を設定できます。

C のインスタンス メソッドの中にある次のようなコーディングは、COBOL Checker によってコンパイル時に拒否されます。

```
move "!" to instItemA
```

クラス C のプログラマは、"setInData" メッセージを自身に送ることによってのみ、instItemA をセットできます。たとえば、次のとおりです。

```
invoke self "setInDataA" using "!"
```

これは、実行時にメッセージ "setInData" をインスタンス オブジェクト C1 に送ります。メッセージはクラス A に到達するまで継承チェーンの上に向かって渡されます。オブジェクト参照 SELF は、メッセージ "setInData" が発生したオブジェクトを常に参照する予約名です。

12.4 クラス オブジェクト動作のプログラミング

ここでは、クラス オブジェクト動作のプログラミングについて説明します。

12.4.1 クラス オブジェクト プログラム

クラス オブジェクト プログラムは、Object COBOL クラス内部に入れ子され、CLASS-OBJECT と END-CLASS OBJECT 見出しによってかっこで囲まれます。クラス オブジェクト プログラムはクラス オブジェクトの動作をまとめ、クラスがロードされたとき、実行時にだけ存在します。

すべてのクラスが クラス オブジェクト プログラムを含むのではなく、実行時のクラス オブジェクトの主な機能は新規インスタンスを作成することで、スーパークラスから継承されます。独自のクラス オブジェクトを記述する主な理由は次のとおりです。

- 初期化コードを含むインスタンス作成メソッドを使用する
- クラスのすべてのインスタンス全体にデータを保存する (メッセージをクラスに送ることによって、インスタンスだけがこのデータにアクセスできる)

クラス オブジェクトの例

この例は、クラス オブジェクト コードの位置を示します。

```
 class-id. Example inherits

         from Base.

* 環境部と Class-control 節

 ...

* 共有データ
```

```
...

class-object                *> クラス オブジェクトのはじめ


    • クラスのデータとコード


...

end class-object.           *> クラス オブジェクトの終わり
```

*** インスタンス オブジェクト コード**

```
...

end class Example.
```

関連項目

- 2.2 クラス
- 第 12 章: クラス プログラム
- 12.2.1 クラス プログラムの構造
- 12.4.1 クラス オブジェクト プログラム
- 12.4.2 クラス データ
- 12.4.3 クラスの初期化
- 12.4.4 クラス メソッド

12.4.2 クラス データ

クラス オブジェクトの Object-Storage 節でクラス オブジェクト データを宣言します。クラス オブジェクト デー

タは、クラスメソッドからのみアクセスでき、サブクラスによって任意で継承できます。

クラスがアプリケーションがロードされると、Object COBOL RTS は Object-Storage 節で宣言されたデータとスーパークラスで宣言されたデータに、メモリ領域を割り当てます。しかし、クラス オブジェクトは、自身を宣言したか、継承したデータに直接アクセスできるだけです。

各クラス オブジェクトは、そのスーパークラスの中で宣言された、自身の固有なデータのコピーを持っています。データ付きで継承するかどうかは関係ありません。これは、一部の他のオブジェクト モデルと対照的です。他のオブジェクトモデルでは、各クラス オブジェクトがスーパークラスと継承されたデータを共有します。図 12-6 は A、B、C の 3 つのクラスを示します。これらはクラス オブジェクト データ項目と、各クラス オブジェクトの実行時に割り当てられたクラス オブジェクト データを宣言します。

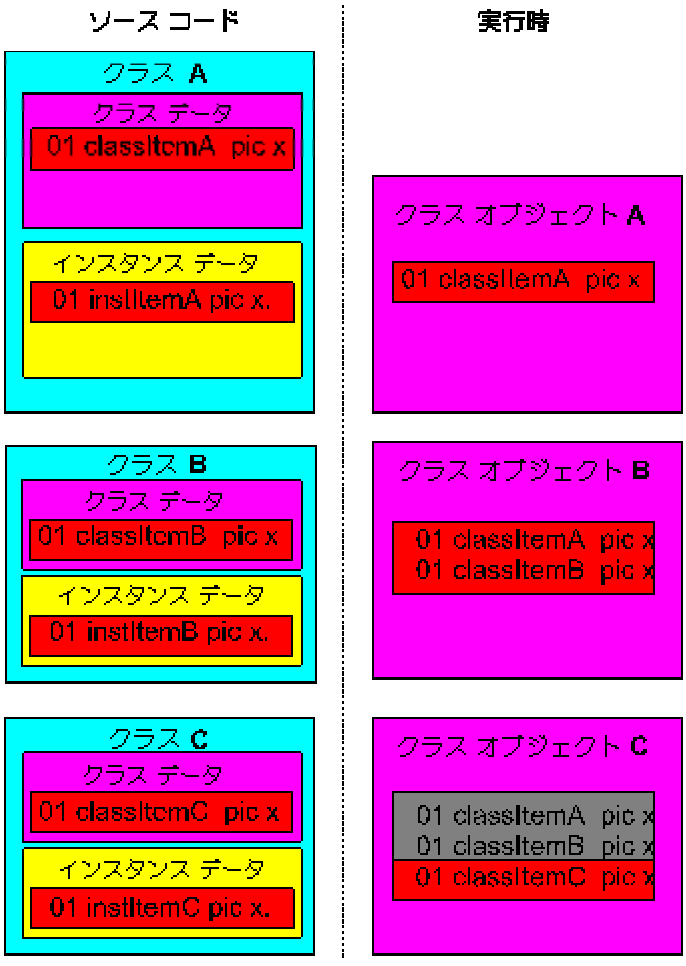


図 12-6 ソースコードおよび実行時のクラスデータ

クラス A　　クラス B はインスタンス データの 1 つの追加項目を宣言し、クラス A からデータ付き (WITH DATA) で継承します。

クラス B　　クラス B はインスタンス データの 1 つの追加項目を宣言し、クラス A からデータ付き

(WITH DATA) で継承します。

クラス C　　クラス C はインスタンス データの 1 つの追加項目を宣言し、クラス B からデータなし (WITHOUT DATA) で継承します。

クラス オブジェクト A　　実行時に、クラス オブジェクト A が単一のデータ項目に記憶域が割り当てられます。

クラス オブジェクト B　　実行時に、クラス オブジェクト B はクラス A で宣言されたデータ項目だけではなく、それ自体に宣言されたデータに記憶域が割り当てられます。クラス B のプログラマは直接、classItemA にアクセスすることができます。クラスはデータ付きで (WITH DATA) 継承するからです。

クラス オブジェクト C　　実行時に、クラス オブジェクト C は classItemC に対して記憶域を割り当て、また classItemA および classItemB に対しても割り当てます。これらはスーパークラスで宣言されます。しかし、クラス C のプログラマはこのデータに直接、アクセスすることはできません。クラス C はデータ付き (WITH DATA) で継承しないからです。クラス オブジェクト C は、クラス A および B のメソッドを通して継承されたデータにのみアクセスすることができます。SELF または SUPER にメッセージを送信することによって、これらのメソッドを起動します。メソッドは継承を通じて見つけられますが、クラス オブジェクト C の固有のデータ空間上で実行されます。

クラス データの例

この例は、クラス データの宣言を示します。

```
class-id. Example inherits

          from Base.

* 環境部と Class-control 節

 ...

* 共有データ

 ...

* クラス オブジェクト コード

 class-object.

 object-storage section.       *> クラス データのはじめ

 01 classDataItem1    pic x.   *> クラス メソッドにだけアクセス可能
```

```
 ...                                *>
```

- **クラス メソッドが続きます**

```
...
 end class-object.
```

*** インスタンス オブジェクト コード**

```
end class Example.
```

関連項目

- 12.3.2 データの継承方法
- 12.4.1 クラス オブジェクト プログラム
- 12.4.2 クラス データ
- 12.4.3 クラスの初期化
- 12.4.4 クラス メソッド

12.4.3 クラスの初期化

クラス初期化コードは、RTS によってクラスをロードしたときに実行されます。このコードは、アプリケーションの実行中に 1 回だけ実行されます。クラスの初期化コードは任意で、ほとんどのクラスは必要としません。

クラス オブジェクト データの初期化のために、クラス初期化コードを使用することができます。たとえば、起動時にクラス オブジェクトが特定のオブジェクトを使用できるように設定したいとします。値の部では静的なデータだけを指定でき、オブジェクトハンドルは実行時に動的に割り当てられるため、クラスの Object-Storage 節の値の部ではこれらを指定することができません。

Object COBOL は、アプリケーションの起動時間を短縮するためにデマンド ローディングのシステムを使用します。つまり、アプリケーション内で、いつ特定のクラスが読み込まれるかを、ユーザが正確に予測できないことを意味します。クラスはアプリケーションの実行中に最初のメッセージを受け取るまで、ロードされ、初期化されることが保証されます。クラスがロードされるタイミングは正確にわからないので、アプリケーションの任意の場所で使用される外部変数の値を設定するために、クラス初期化コードに依存することはできません。

持続機能を使用する場合、クラスの初期化はアプリケーションが初めて実行されたときのみ行われます。その後の

実行ではクラスが初期化されていることを前提とし、初期化コードは実行されません。

クラスの初期化コードの作成方法

1. 手続き部をコード化し、Class-Object 見出しを開始します。
2. 文をコード化します。

   ```
   invoke self "initializeClass"
   ```

3. "initializeClass" のクラスメソッドをコード化します。このメソッドが実行されるのは、クラスが最初に RTS でロードされる場合のみです。

注記 :

NetExpress の今後のリリースでは、RTS は、ロードされる際に "initializeClass" メッセージを自動的に各クラスに送信し、クラスの手続き部は無視されるようになります。クラス初期化コードの作成技法として上記で推奨しているのは、この変更と互換性のある送信です。 何も行わないデフォルトの "initializeClass" メソッドは、Object COBOL メタクラスの一部として提供されます。これは、このメソッドを含まないクラスが、期待どおりに動作し続けるようにするためです。

クラスの初期化例

この例は、クラス初期化コードを示します。

```
class-id. Example inherits
          from Base.
* 環境部と Class-control 節
 ...
* 共有データ
 ...
```

- **クラス オブジェクト コード**

```
method-id. "initializeClass". *> 初期化メソッドのはじめ
procedure division.
    ...                          *> クラスの初期化へのコード
```

```
    exit method.

 end method "initializeClass". *> 初期化メソッドの終わり

...

* インスタンス オブジェクト コード

 ...

 end class Example.
```

関連項目

- 2.2 クラス
- 第 12 章: クラス プログラム
- 12.2.1 クラス プログラムの構造
- 12.4.1 クラス オブジェクト プログラム
- 12.4.2 クラス データ
- 12.4.3 クラスの初期化
- 12.4.4 クラス メソッド

12.4.4 クラス メソッド

クラス メソッドはクラス オブジェクトの中で入れ子されたプログラムです。アプリケーションを実行するときに、クラス オブジェクトの特定の型のオカレンスが1つだけあります。したがって、クラス メソッドは一般にインスタンス オブジェクトの作成と管理を考慮します。

クラス メソッドは、クラス オブジェクト データとクラス プログラム データ部で宣言されたすべての共有データにアクセスできます。

クラスメソッドとインスタンスメソッドをコーディングする方法は同じです。違いは、クラス プログラムのソースコード内の位置と、アクセスできるデータの範囲だけです。

クラス メソッドの例

この例は、クラス メソッドの位置を示します。

```
class-id. Example inherits

          from Base.
```

```
* 環境部と Class-control 節

 ...

* 共有データ

 ...
```

- **クラス オブジェクト コード**

```
class-object.
```

- **クラス データ**

```
 ...
 method-id. "newWithData".
```

```
* メソッド コードとデータ

 ...
 end method "newWithData".
```

- **さらにメソッドが続きます**

```
...
 end-class object.
```

```
* インスタンス オブジェクト コード

 ...
 end class Example.
```

関連項目

- 2.3 メソッド

- 12.4.1 クラス オブジェクト プログラム
- 12.4.2 クラス データ
- 12.4.3 クラスの初期化
- 12.4.4 クラス メソッド
- 12.5.4 インスタンス メソッド
- 12.6.1 メソッド プログラム

12.5 インスタンス オブジェクト動作のプログラミング

ここでは、インスタンス オブジェクト動作のプログラミングについて説明します。

12.5.1 インスタンス オブジェクト

Object プログラムは OBJECT COBOL クラスの中で入れ子され、OBJECT 見出しと END-OBJECT 見出しで囲まれます。Object プログラムは、クラスのインスタンスの動作を具体的に指示します。

インスタンス オブジェクトの例

この例は Object プログラムの位置を示します。これはインスタンス オブジェクトを記述しています。

```
class-id. Example inherits
          from Base.
* 環境部と Class-control 節
 ...
* 共有データ
 ...
* クラス オブジェクト コード
 ...
* インスタンス オブジェクト コード
 object.                        *> インスタンス オブジェクトのはじめ
```

- **インスタンスのデータとコード**

```
...

 end object.                    *> インスタンス オブジェクトの終わり

end class Example.
```

関連項目

- 2.1 オブジェクト
- 第 11 章: プログラムでのオブジェクトの使い方
- 12.5.1 インスタンス オブジェクト
- 12.5.2 インスタンス データ
- 12.5.3 インスタンスの初期化
- 12.5.4 インスタンス メソッド

12.5.2 インスタンス データ

インスタンス データは、インスタンス オブジェクト プログラムの Object-Storage 節で宣言します。インスタンス データはインスタンス メソッドからだけアクセスでき、各インスタンスは自身のインスタンス データだけを見ることができます。インスタンス データは、サブクラスのインスタンスによっても、任意に継承することができます。

インスタンスが作成されると、Object COBOL RTS は Object-Storage 節で宣言されたデータとスーパークラスで宣言されたインスタンス データのためのメモリの領域を割り当てます。しかし、インスタンス オブジェクトが直接アクセスできるデータは、自身を宣言したか、継承したデータだけです。インスタンス オブジェクトが、継承しなかったデータにアクセスするには、スーパークラスにメッセージを送る必要があります。

実際にはオブジェクトの属性の一部であるデータを、インスタンス データとして宣言するだけです。一時的な作業または計算のために必要なデータ項目を宣言する必要があります。たとえば、一時データを使用するメソッドやクラス プログラムのはじめの Working-Storage 節などです。

このような Working-Storage 節の使い方は、Micro Focus の独自の機能で、OO COBOL に関する ANSI 標準規格には含まれていません。

たとえば、本書の Object COBOL チュートリアルで使用する Account オブジェクトのために宣言されたインスタンス データは、顧客名、口座番号、残高だけです。すべてのインスタンスについて同じデータ (たとえば、利率) がある場合、クラス オブジェクト データとして宣言すべきです。

これは、アプリケーションが新しいインスタンス オブジェクトまたはクラスのサブクラスの新しいインスタンスを作成するたびにインスタンスデータとして宣言した記憶域の各バイトが割り当てられるからです。厳密にオブジェクトの属性であるデータだけをインスタンスデータとして宣言することによって、メモリオーバーヘッドを最小限に抑えられます。

インスタンス データの例

この例はインスタンス データの宣言を示します。作成される各新規オブジェクトはインスタンス データのメモリの領域が割り当てられます。

```
class-id. Example inherits

          from Base.

* 環境部と Class-control 節

 ...

* 共有データ

 ...

* 初期化コード

 ...
```

- **クラス オブジェクト コード**

```
...

* インスタンス オブジェクト コード

 object.                    *> インスタンス オブジェクトのはじめ

 object-storage section.      *> インスタンス データのはじめ

 01 aDataItem1       pic x.   *> インスタンスにだけアクセス可能

 ...
```

- **インスタンス メソッドが続きます**

```
...
end object.                    *> インスタンス オブジェクトの終わり
...
end class Example.
```

関連項目

- 2.1 オブジェクト
- 第 11 章: プログラムでのオブジェクトの使い方
- 12.5.1 インスタンス オブジェクト
- 12.5.2 インスタンス データ
- 12.5.3 インスタンスの初期化
- 12.5.4 インスタンス メソッド

12.5.3 インスタンスの初期化

オブジェクトの初期化をどのようにコーディングするかは、おもにプログラマが決めることができます。インスタンスの初期化には、クラスの初期化のような明示的な機構はありません。最も簡単で読みやすい方法は、メソッド "initialize" をコーディングし、インスタンスを作成した後にメソッド "initialize" を起動するインスタンス作成メソッド (常にクラスメソッドである) を作成することです。

作成するメッセージと共に提供されるデータによって、初期化されるオブジェクトもあります。作成メソッドは、これらのパラメータをメソッド "initialize" に渡す必要があります。他の型のオブジェクトは、常に作成時と同じ状態に初期化されます。

一般にウィンドウとダイアログボックスは、このカテゴリに当てはまります。初期化コードは、毎回同じオブジェクトとラベルのセットによる描画からできているのが普通です。特定のクラスのすべてのインスタンスによって使用される初期化データを宣言するのに最もよい場所は、クラスの最初の Working-Storage 節です。

希望するデータを、Working-Storage 項目のための value 句に入れることができます。データ項目は、オブジェクト インスタンスすべてによって名前で直接アクセスでき、データ領域はクラスのロード時に一度だけ割り当てられます。

このような working-storage 節の使い方は、Micro Focus Object COBOL 独自の機能で、OO COBOLに関するANSI標

準規格に現在は含まれていません。

インスタンス作成メソッドの例

この例は新規インスタンスを作成し、一部のデータでそれを初期化するための作成メソッドを実現するクラスを示します。

```
class-id. A data is protected

          inherits from Base.


 object section.

 ...

 method-id. "newWithData"

 linkage section.

    01 lnkObject     object reference.

    01 lnkName       pic x(80).


 procedure division using lnkName

              returning lnkObject.


*----Base からメソッド "new" を使用して A の新規インスタンスを作成

    invoke super "new" returning lnkObject

*----初期化メッセージを送る

    invoke lnkObject "initialize" using lnkName

    exit method.

 end method "newWithData".

 object.

 object-storage section.
```

```
    01  theName    pic x(80).
    ...

method-id. "initialize"

linkage section.
01  lnkName    pic x(80).

procedure division using lnkName.
*----オブジェクトのインスタンス データ中に初期化パラメータを格納する
*
    move lnkName to theName
    exit method.
end method "setInItemA".
...

end object.
end class "A".
```

クラス A の "newWithData" メソッドは SUPER に対して "new" を送信します。オブジェクトが SUPER に対してメッセージを送信するとき、それ自体にメッセージを送信します。Object COBOL RTS はメッセージを送信するクラスのコードではなく、スーパークラスのコードで一致するメソッドを探し始める点だけが異なります。

関連項目

- 2.3 メソッド
- 12.4.4 クラス メソッド
- 12.5.1 インスタンス オブジェクト

- 12.5.2 インスタンス データ
- 12.5.3 インスタンスの初期化
- 12.5.4 インスタンス メソッド
- 12.6.1 メソッド プログラム
- 12.6.4 新規インスタンスの作成方法

12.5.4 インスタンス メソッド

インスタンス メソッドは、インスタンス オブジェクト プログラムの内側で入れ子されます。インスタンス メソッドは、アプリケーションがオブジェクトの状態を変更し、その属性をクエリーする方法です。

インスタンス メソッドは、インスタンス データとクラス プログラム データ部で宣言された共有データすべてにアクセスできます。

クラスとインスタンス メソッドは同様の方法でコーディングします。唯一の相違点はクラス プログラム ソース コードの位置とアクセスするデータの範囲です。

インスタンス メソッドの例

次に、クラスの中のインスタンス メソッドの位置を示します。

```
class-id. Example inherits
          from Base.
```

*** 環境部と** `Class-control` **節**

```
 ...
```

*** 共有データ**

```
 ...
```

*** クラス オブジェクト コード**

```
 ...
```

*** インスタンス オブジェクト コード**

```
 object.
```

- **インスタンス データ**

```
...
method-id. "setItem1".
```

*** メソッド コードとデータ**

```
...
end method "setItem1".
```

- **さらにメソッドが続きます**

```
...
end object.

end class Example.
```

関連項目

- 2.3 メソッド
- 12.4.4 クラス メソッド
- 12.5.1 インスタンス オブジェクト
- 12.5.2 インスタンス データ
- 12.5.3 インスタンスの初期化
- 12.5.4 インスタンス メソッド
- 12.6.1 メソッド プログラム

12.6 プログラミング メソッド

ここでは、プログラミング メソッドについて説明します。

12.6.1 メソッド プログラム

各メソッドは、METHOD-ID および END METHOD 見出しで囲まれる入れ子プログラムです。METHOD-ID 見出し名はメソッドです。メソッドはメソッドを含むオブジェクトがメソッド名と一致するメッセージに送信されるたびに起動されます。

クラス メソッドはクラス オブジェクト プログラムの中に表示されます。インスタンス メソッドはオブジェクト プログラム自身の中に表示され (OBJECT と END OBJECT 見出しの間) 、その後に Object-Storage 節が続きます。

メソッドの例

この例はメソッド "setInItemA" を示します。これは 1 文字を記憶し、それを大文字に変換します。このメソッドは、文字がすでに大文字かゼロの場合には 1 を返します。データ項目 inItemA を参照します。これは、Object-Storage で宣言されていることを前提とします。

メソッドはすべての一時作業データに対して Local-Storage を使用します。メソッドが起動されるたびに、Local-Storage データの新しい割り当てを取得します。実行が EXIT METHOD 文のメソッドを終了すると、割り当て解除されます。

パラメータは Linkage 節を通してメソッドと受け渡しされます。

```
method-id. "setInItemA".

local-storage section.
01  aTempItem       pic x(4) comp-5.

linkage section.
01  lnkValue        pic x.
01  aResult         pic x comp-5.
  88 isUpperCase     value 1.
  88 isLowerCase     value 0.
```

```
 procedure division using lnkValue
                returning aResult.
     move lnkValue to inItemA
     move 1 to aTempItem
     call "CBL_TOUPPER" using inItemA
                           aTempItem
     if lnkValue = inItemA
        set isUpperCase to true
     else
        set isLowerCase to true
     exit method.
  end method "setInItemA".
```

関連項目

- 2.3 メソッド
- 12.4.4 クラス メソッド
- 12.5.4 インスタンス メソッド
- 12.6.1 メソッド プログラム
- 12.6.2 メソッド データとパラメータ
- 12.6.3 自分宛てのメッセージ送信
- 12.6.4 新規インスタンスの作成方法

12.6.2 メソッド データとパラメータ

クラス メソッドはクラスの Object-Storage データにアクセスでき、インスタンス メソッドはインスタンスの Object-Storage データにアクセスできます。その他の面では、クラスとインスタンス メソッドは外見もメソッドも同じです。いずれのメソッドも、クラス プログラムのデータ部で宣言された共有データにアクセスすることができます。

メソッドは自身のデータをもつこともでき、メソッドのデータ節で次の型を宣言できます。

- Working-Storage

  すべてのメソッドの起動と、すべてのオブジェクト インスタンスに共通のデータ項目。

- Local-Storage

  現在のメソッドの起動にローカルなデータ項目。

- Linkage

  メソッドとの間でパラメータを受け渡しするためのデータ項目。

RTS はメソッドが起動されるたびに Local-Storage にメモリを割り当て、メソッドの終了後に割り当てを解除します。これには、再帰をサポートしているという利点があります。メソッドは、自分自身を直接起動しない場合も再帰的な場合があります。つまり、もとのメソッドを2回目に起動する別のオブジェクトの中でメソッドを起動する場合です。

working-storageはメソッド内の一時変数のために使用できますが、永久的に割り当てられ、同じクラスのすべてのオブジェクトで起動されたすべてのメソッドによって共有されます。したがって、再帰的に起動するメソッドには適していません。

たとえば、オブジェクト A1 (クラス A のインスタンス) にあるメソッド "setData" は、オブジェクト B1 のメソッド "getDetails" を起動します。オブジェクト B1 にあるメソッドは、オブジェクト A2 (やはりクラスAのインスタンス) の "setData" を起動します。最初に起動した "setData" (実行がまだ終了していない) の Working-Storage にあったデータは、2 回目の起動によって上書きされます。 "setData" の 2 回の起動のための Local-Storage データは、完全に別々になっています。

Linkage 節は、メソッドと受け渡しされるすべてのパラメータを宣言するために使用されます。一般に、パラメータは参照によって渡されます。つまり、Linkage 節の項目には実際のメモリが対して割り当てられません。

注記 :

Local-Storage データは Micro Focus の独自の機能です。OO COBOL の ANSI 標準の提案では、Micro Focus Local-Storageと等しいメソッドの中で宣言されるときに Working-Storage を定義します。コンパイラ設定、OOCTRL(+W) を設定することによって、Micro Focus Object COBOL でメソッドの中の Local-Storage を指すために Working-Storage を使用することができます。これを使用する場合、すべてのインスタンスに静的なデータを宣言するので、メソッド Working-Storage を使用できなくなります。

例については、前項の「メソッドの例」を参照してください。

関連項目

- 2.3 メソッド

- 12.4.4 クラス メソッド
- 12.5.4 インスタンス メソッド
- 12.6.1 メソッド プログラム
- 12.6.2 メソッド データとパラメータ
- 12.6.3 自分宛てのメッセージ送信
- 12.6.4 新規インスタンスの作成方法

12.6.3 自分宛てのメッセージ送信

クラスやインスタンス オブジェクトが自分自身にメッセージを送りたいことがよくあります。これには、2つの大きな理由があります。

- 複数の異なるメソッドによって使用されるコードを、自分自身のメソッド内に入れることができます。このメソッドは、サブルーチンのように使用できます。
- スーパークラスで宣言され、オブジェクトが継承していないデータ項目の値を獲得または変更したいことがあります。これには、そのためのメソッドがスーパークラスになければなりません。これは、メソッドは常にサブクラスによって継承されるからです。オブジェクトは、継承されたメソッドを呼び出し、自分自身のデータに適用するために、自分自身にメッセージを送ります。

クラス プログラムで使用でき、オブジェクト作成時に Object COBOL RTS によって初期化される 3 つの予約済みのオブジェクト参照があります。

- SELF
- SELFCLASS
- SUPER

パラメータとしてほかのメソッドに渡すことができるオブジェクト参照データとして、SELF または SELFCLASS を使用することもできます。INVOKE 文のターゲット オブジェクトとしてだけ、SUPER を使用することができます。

自分宛てのメッセージ送信例

メッセージを送信するためには、メッセージの受信者とメッセージセレクタを指定します。オブジェクトはそれ自体、またはそれ自体のクラスにメッセージを送信することができます。この例は、メッセージ自身に送信するオブジェクトの例です。

```
invoke self "getInstanceAttrib1" returning someData
```

オブジェクトは、スーパークラスで定義されているメソッドを再実現する場合もありますが、メソッドのスーパークラス バージョンに引き続きアクセスする必要があります。これは、メッセージを自分自身に送信するオブジェクトの例です。この場合、スーパークラスで実行メソッドを見つける必要があります。

```
invoke super "getInstanceAttrib1" returning someData
```

オブジェクトはクラスにメッセージを送信したいことを指定することもできます。

```
invoke selfclass "getClassAttrib1" returning someData
```

関連項目

- 2.6 メッセージと結合
- 11.3 メッセージの送信
- 12.6.1 メソッド プログラム
- 12.6.2 メソッド データとパラメータ
- 12.6.3 自分宛てのメッセージ送信
- 12.6.4 新規インスタンスの作成方法

12.6.4 新規インスタンスの作成方法

クラスからインスタンス オブジェクトを作成できるようにする必要があります。簡単に言うと、新しいオブジェクトインスタンスを作成することは、そのデータとそれを参照するオブジェクトハンドルのためのメモリを割り当てることを意味します。基本的なタスクは、Base にある "new" というクラスメソッドによって取り扱われます。

インスタンスに関する初期化を行いたくない場合、作成のためのコーディングはクラスの中では必要ありません。単純に "new" メソッドを継承できます。

多くの場合、作成したオブジェクトについて何らかのインスタンス初期化を行います。その場合、オブジェクト作成のためのクラスメソッドを自分でコーディングする必要があります。しかし、独自のインスタンス作成メソッドでは、オブジェクト作成とメモリ割り当てのための基本的な機構として常にメタクラス メソッド "new" を使用します。

メソッドを作成しないクラスの例

この例は、インスタンス作成のために固有のコードを提供するクラスを概説するのではなく、メタクラス Behavior から継承する "new" メソッドを使用します。

```
 class-id. simple inherits from base.

* simple のクラス メソッドはない

 ...

 object.

 object-storage section.

    01   simpleInstanceData   pic x(80).

* simple のインスタンス メソッドはここでは示さない

 ...

 end object.

 end class simple.
```

アプリケーションが Simple の新しいインスタンスを必要とする場合は、次のようにコーディングしてください。

```
invoke simple "new" returning anObject
```

ただし、anObject は OBJECT REFERENCE 型のデータ項目です。継承された "new" メソッドは Simple のインスタンスに必要な 80 バイトを割り当て、オブジェクト ハンドルを anObject に入れます。

インスタンス作成メソッドの例

この例は新規インスタンスを作成し、一部のデータでそれを初期化するための作成メソッドを実現するクラスを示します。

```
class-id. A data is protected

         inherits from Base.


 object section.

 ...

 method-id. "newWithData"

 linkage section.

    01 lnkObject     object reference.
```

```
    01 lnkName       pic x(80).

 procedure division using lnkName
              returning lnkObject.

*----Base からメソッド "new" を使用して A の新規インスタンスを作成
    invoke super "new" returning lnkObject
*----初期化メッセージを送る
    invoke lnkObject "initialize" using lnkName
    exit method.
 end method "newWithData".
 object.
 object-storage section.
    01  theName    pic x(80).
    ...

 method-id. "initialize"

 linkage section.
 01  lnkName    pic x(80).

 procedure division using lnkName.
*----オブジェクトのインスタンス データ中に初期化パラメータを格納する
*
    move lnkName to theName
```

```
    exit method.

end method "setInItemA".

...


end object.

end class "A".
```

クラス A にある "newWithData" メソッドは、SUPER に対してメッセージ "new" を送ります。オブジェクトが SUPER に対してメッセージを送るとき、自身にもメッセージを送信します。Object COBOL RTS はメッセージを送るクラスのコードではなく、スーパークラスのコードで一致するメソッドを探し始める点だけが異なります。

関連項目

- 12.5.3 インスタンスの初期化
- 12.6.1 メソッド プログラム
- 12.6.2 メソッド データとパラメータ
- 12.6.3 自分宛てのメッセージ送信
- 12.6.4 新規インスタンスの作成方法

第13章 クラス ライブラリの紹介

Object COBOLには、プログラマが使用したりサブクラス化できる定義済みクラスのクラス ライブラリが付属しています。この章では、Object COBOLのクラス ライブラリを紹介し、その使い方について説明します。

13.1 概要

Object COBOLのクラス ライブラリは、プログラマが独自のアプリケーションをすばやく作成するのに役立つクラスのセットを提供します。NetExpress には、以下の2つのクラス ライブラリがあります。

- 基本クラス ライブラリ
- GUI クラス ライブラリ

基本クラス ライブラリ

基本クラス ライブラリは、オブジェクト指向プログラムを記述するユーザをサポートします。独自のObject COBOLクラスを記述したいユーザは、コピーや破棄 (仕上げ) のように一部の基本的なオブジェクト操作を提供するクラスBase から継承することができます。

基本クラス ライブラリはまた、次のクラスも提供します。

- システムクラス
  基本的なシステムサポートを提供するBaseやBehaviorなどのクラス
- 固有クラス
  COBOLデータ項目にメッセージを送るためのクラス。第15章「固有データ」参照。
- 例外処理クラス
  オブジェクトで発生した例外を処理するクラス。第17章「例外取り扱いのフレームワーク」参照。
- コレクションクラス
  オブジェクトのコレクションを管理するオブジェクトを作成するためのクラス。OOプログラミングで特に便利です。第14章「コレクションのフレームワーク」参照。

コレクション クラスは特に豊富にあります。これらは従来の COBOL で問題となりうる領域外では非常に苦労します。たとえば、配列の大きさが分からないときにデータ配列に注意する場合などです。

OrderedCollection オブジェクトはいくつもの要素を管理することができ、それらが追加されたときに特別な要素を調整するよう拡大します。基本クラス ライブラリには、Set (セット)、Bag、Array、OrderedCollection、SortedCollection、および Dictionary (ディクショナリ) クラスが含まれます。

GUI クラス ライブラリ

GUI クラス ライブラリは、Windows 95 (および Windows NT) のネイティブ インタフェースを利用するグラフィック アプリケーションの記述をサポートします。これには、ダイアログ ボックス、入力フィールド、プッシュボタン、リスト ボックスやツリー表示など多くの共通制御に対するクラスが含まれます。

NetExpress の GUI クラス ライブラリは全体がObject COBOLで記述されており、Windows API を呼び出すレベルで書かれています。これにより以下の 2 つの利点があります。

- 拡張性
  クラス ライブラリでまだ利用できない Windows 機能を使用したい場合で Windows API 呼び出しをコーディングするための専門知識がある場合は、クラス ライブラリ オブジェクトをサブクラス化しそれらに拡張することができます。既存のソース コードを参照できる利点もあり、その結果 Micro Focus コードが API と対話する方法を参照できます。

- 透過性
  必要な場合は、アプリケーションをデバッグする時のクラス ライブラリ コードを入れておくことができます。デバッグできない、あるいは第 2 言語を使用してデバッグする必要がある、ブラック ボックス内の関数を呼び出すことはありません。起動されるすべてのコードは、Windows API を呼び出すポイントまではObject COBOLです。

オブジェクト指向は、アプリケーションのエンドユーザから見たオブジェクトとプログラマが記述するオブジェクトとのマッピングを使って、グラフィカル インタフェースを記述するのに非常に一般的な方法です。GUI クラス ライブラリの仲間として、NetExpress には GUI アプリケーション テンプレートがあります。これは、GUI アプリケーションを記述するためのクラスのスタータ セットを提供し、これを変更したり拡張してクラスを作成できます。

- GUIクラス
  グラフィカルインタフェースを作成および管理するためのクラス。第18章「GUIクラスのフレームワーク」参照。

- GUI アプリケーション テンプレート
  25.3「GUIアプリケーションのテンプレート」参照。

クラス ライブラリには、COBOLファイルを取り扱うためのクラスはまだ含まれていません。Object COBOLは、完全なCOBOLファイルの取り扱い構文をサポートしており、これをクラスやプログラムの中で使ってCOBOLファイル型の読み書きができます。

クラス ライブラリのドキュメンテーションでは、クラス ライブラリのプライベートな部分を参照している個所があります。プライベートなことが記述されている個所は、文の中にそのことが示されています。プライベートクラスからサブクラス化したり、プライベートメソッドをObject COBOLアプリケーションの中で使用しないでください。ドキュメンテーションの中にプライベート(PRIVATE)と記されたものがないかぎり、パブリックインタフェースの一部と見なすことができます。パブリックとプライベートについては次の各項で説明します。

本書とともに、Object COBOLのクラス ライブラリ用のソースコードを用意しました。これは、次の理由によります。

- Object COBOLプログラミングの追加例として
- クラス ライブラリの動作を理解するため
- クラス ライブラリの再コンパイルとアニメートを可能にするため（独自のアプリケーションをデバッグするときに便利な場合があります）
- 必要に応じてクラス ライブラリを変更できるようにするため（「クラス ライブラリのアニメート」の項にある警告を参照）

13.1.1 パブリックインタフェース

クラス ライブラリへのパブリックインタフェースは、プログラマがObject COBOLアプリケーションを作成するのに使うためのクラス ライブラリ内のメソッドとクラスからできています。

13.1.2 プライベートインタフェース

クラス ライブラリのプライベートインタフェースは、クラス ライブラリを使うプログラマには使用されないメソッドとクラスからできています。プライベートインタフェースは、クラス ライブラリの実現の一部分であり、将来のリリースで変更されることがあります。

プライベートメソッドは、クラス ライブラリのソースコード内の注釈などによって、そのことが示されています。

プライベートインタフェースには、クラス ライブラリ オブジェクトにあるすべてのクラスとインスタンスデータも入っています。このデータはどのサブクラスでも直接使用しないでください。このデータにアクセスするには、スーパークラス内のメソッドを使用してください。

13.2 クラス ライブラリの記述

クラス ライブラリの記述は、次の2つに分けられます。

- フレームワーク
- クラスへのパブリックインタフェース

フレームワークについては本書に記述されています。オブジェクトへのパブリックインタフェースは、オンラインの『Object COBOL クラス ライブラリ リファレンス』に記述され、クラス ライブラリ内の各クラスについて次の情報が提供されます。

- クラスの説明
- 各パブリッククラスメソッドの記述（継承されるメソッドを含む。Baseから継承されるメソッドは除く）

- 各パブリックインスタンスメソッドの記述（継承されるメソッドを含む。Baseから継承されるメソッドは除く）

プライベートメソッドについては記述されていません。またBaseから継承されるメソッドは、明示的に無視されている場合を除き、クラス ライブラリのすべてのオブジェクトが継承するため記述されていません。このためすべてのオブジェクトに共通のメソッドは、Baseクラスに記述されています。

13.3 フレームワーク

フレームワークは、各種のオブジェクトが互いに対話するための手段です。個々のクラスオブジェクトとインスタンスオブジェクトは、そのオブジェクトと通信するための方法を記述したインタフェースを持っています。

フレームワークは、特定の機能を実現するために共に動作する、複数の異なる種類のオブジェクトによって使用されるプロトコルです。この部の他の章では、クラス ライブラリで使用されるおもなフレームワークについて説明しています。この情報は、クラス ライブラリを効率的に使用するのに役立ちます。

関連項目

- 第14章「コレクションのフレームワーク」
- 第15章「固有データ」
- 第16章「Callbackのフレームワーク」
- 第17章「例外取り扱いのフレームワーク」
- 第18章「GUIクラスのフレームワーク」
- 第19章「コンポーネントのフレームワーク」

13.4 クラス ライブラリのアニメート

クラス ライブラリの各部をアニメートすると、その動作を理解したり、自分のアプリケーションのバグを追跡するのに役立ちます。アプリケーションからは、一度に1つか2つのクラス ライブラリ プログラムだけをアニメートしてみることを強くお勧めします。

Object COBOL自身は、クラス ライブラリを起動時に使用します。したがって、クラス ライブラリ全体をアニメートしようとすると、アプリケーションがロードされる前に意図しないコードに入ってしまいます。

- クラス ライブラリの各部をアニメートするには

  1. オブジェクトツールグループのClass Browserのアイコンをダブルクリックします。ファイルリストからMfclass.prjファイルを選択します。

2. 「オプション」メニューから「チェック」を選択し、チェックプログラムのコマンド行を次のように変更します。

cobol % omf "gnt";

3. 「COBOL」メニューから「すべてチェック」を選択します。
クラス ライブラリにあるすべてのプログラムについて、固有コード.gntファイルが作成されます。

4. アニメートしたいクラスの.gntファイルを削除します。

5. 「オプション」メニューから「チェック」を選択し、チェックプログラムのコマンド行を次のように変更します。

cobol % editor(mf) anim nognt mfoo;

6. アニメートしたいクラスを選択し、「COBOL」メニューから「チェック」を選択します。アニメートしたいすべてのクラスについて、このステップを繰り返します。

7. OOSW環境変数を設定します。

set oosw= -l

このスイッチは、OO RTSに対してclass.lbrをロードせず、いまコンパイルした個々のクラス ライブラリを使用するように指示します。クラス ライブラリのライブラリは、COBDIRで設定されたパスか、アプリケーションを実行する現在のディレクトリに入っていなければなりません。

8. アプリケーションを実行します。

警告：クラス ライブラリのソースコードは変更しないでください。実行に関する詳細はプライベートと考えられており、Object COBOLのリリースによって異なります。Micro Focus社では、このような変更を禁止していませんが、将来のリリースでのサポートは保証できません。

第14章 コレクションのフレームワーク

この章では、Object COBOLのコレクションクラスによって使用されるフレームワークを紹介します。コレクションのフレームワークでは、コレクションとそこに格納される要素の間の通信をおもに取り扱います。

14.1 概要

Object COBOLのコレクションクラスでは、その要素をオブジェクトであるとみなします。コレクションは、その要素のサイズや内容などの情報を得るために、要素にメッセージを送ります。（オブジェクトではなく）固有データを格納する場合も、コレクションは固有クラスで実現されたメソッドを使用し、INVOKE...AS文を使ってデータにメッセージを送ります。

コレクションに格納する要素には、メソッドがメッセージに対して応答するためのインタフェースを含んでいる必要があります。これらの大部分のデフォルトバージョンはBaseクラスに用意されていますが、特定のオブジェクトについて、これらを無視し、別の動作またはより効率的な動作を与えることもできます。

コレクションクラスのソートメソッドや繰返しメソッドに専用のインタフェースに準拠したメソッドを提供する必要もあります。

14.2 コレクションの分類

クラス ライブラリにある各種のコレクションは、次の属性によって分類することができます。

- 索引付きまたは索引なし
  索引付きコレクションの各要素は、その位置を与えることによってアクセスできます。これは配列を使用する場合と同じです。索引なしコレクションでは、要素を格納するために決められた順序はありません。

- 自動成長または手動成長
  自動成長のコレクションは、コレクションを作成したときの容量を超えると大きくなります。手動成長のコレクションは、メッセージ"grow" が送られたときだけ大きくなります。

- 重複許可または重複禁止
  コレクションによっては項目の重複が禁止されているときがあります。この場合、コレクションにすでにあるのと同じ値の項目を追加しようとすると、例外が起こります。

使用可能なコレクションクラスとその属性の一覧を次に示します。

Bag	索引なし、自動成長、重複許可
Array	索引付き、手動成長、重複許可

CharacterArray	索引付き、手動成長、重複許可
OrderedCollection	挿入順による索引付き、自動成長、重複許可
SortedCollection	ソート順による索引付き、自動成長、重複許可
ValueSet	索引なし、自動成長、重複値禁止
IdentitySet	索引なし、自動成長、重複オブジェクトハンドル禁止
Dictionary	キーによる索引付き、自動成長、重複キー値禁止
IdentityDictionary	キーによる索引付き、自動成長、重複キーオブジェクトハンドル禁止

ValueSet と IdentitySet は、重複項目を決定する方法だけが異なります。ValueSet では項目の値を比較し、重複値を禁止します。IdentitySet ではオブジェクト参照を比較し、同じオブジェクトの複数の格納を禁止します。

Dictionary と IdentityDictionary は、キーとデータの組み合わせを格納する特殊な種類のコレクションです。これらは ValueSet と IdentitySet が重複項目を判断するのと同じ方法で、重複キーを決定します。

14.3 コレクションの作成

オブジェクト (参照のコレクション) または固有データ (値のコレクション) のいずれかを格納するコレクションを作成することができます。参照のコレクションには、2 種類以上のオブジェクトを格納することができます。一方、値のコレクションは、特定の種類と長さを持つデータを格納するように初期設定されています。

コレクションを作成する前に、どちらのタイプを作成するかを決定する必要があります。次の項目を決定する必要があります。

- 索引付きまたは索引なしのどちらにするか
- 重複禁止にするか
- 初期容量を超えたときに自動成長させるか

使用可能なタイプのコレクションは、上の組み合わせで大部分が適用できます。

参照のコレクションを作成する方法

1. 初期のサイズとテンプレートをパラメータとして渡して、作成したいコレクションのクラスに対して"ofReferences"メッセージを送信します。

値のコレクションを作成する方法

1. 固有データ クラスの1つから、固有データのテンプレートのクローンを作成します。

2. 初期のサイズとテンプレートをパラメータとして渡して、作成したいコレクションのクラスに対して"ofValues"メッセージを送信します。

15.2.1「固有データクラスのクローン化」を参照してください。

関連項目

- 14.4「ディクショナリの作成」
- 14.2「コレクションの分類」
- 第15章「固有データ」

14.4 ディクショナリの作成

ディクショナリは、特殊なソートの索引付きコレクションであり、キーとデータの組み合わせ (アソシエートと呼ばれます) を格納します。ディクショナリでは、データを格納または検索するときの索引としてキーを使用します。ディクショナリでは、重複キーを格納することはできません。

他のコレクションタイプと同様に、ディクショナリにはオブジェクトまたは固有データのいずれかを格納することができます。しかし、ディクショナリでは、キー部分またはデータ部分がそれぞれ固有値にもオブジェクトにもなることができます。これにより、固有値またはオブジェクトの記憶域について、次のような組み合わせが可能です。

- キー部分とデータ部分が両方ともオブジェクト
- キーが固有値、データがオブジェクト
- キーがオブジェクト、データが固有値
- キー部分とデータ部分が両方とも固有値

キー部分がオブジェクトである場合、ハッシュおよび等価性の比較のためのメソッドを使用する必要があります。CharacterArray などのオブジェクトには、すでにこれらのメソッドがありますが、キー部分として独自のタイプのオブジェクトを使用する場合は、独自の機構を使用する必要があります 。

ディクショナリを作成する場合、キー部分とデータ部分をどのように格納するかを指定するために、テンプレートを与える必要があります。テンプレートは Association クラス、または Association クラスのクローンのいずれかとなります。

Associationクラスは、データ格納のためのテンプレートの作成に使われる固有データクラスのためのクラスと同様にクローン化可能なクラスです。Association テンプレートは実際には 2 つのテンプレートから構成されています。1 つはキー用のテンプレート、もう 1 つはデータ用のテンプレートです。いずれのタイプのディクショナリを作成する場合にも、Association テンプレートを作成する必要があります。

ディクショナリオブジェクトのためのテンプレートを作成したら、ディクショナリ自身を作成するために使用できる手段として2種類の方法があります。それは、"ofValues" を使う方法と "ofAssociations" を使う方法です。ディクショナリ "ofValues" は、各要素をキーとデータの組み合わせとして格納します。ディクショナリ "ofAssociations" は、各要素をディクショナリの作成に使用した Association テンプレートのインスタンスとして格納します。

ディクショナリ "ofAssociations" では、ディクショナリにある各エントリに対して、実際には次の 3 つの項目が格納されます。

- アソシエート インスタンス
  キーとデータが含まれるオブジェクト
- アソシエート内に含まれているキー
  オブジェクトまたは固有値
- アソシエート内に含まれているデータ
  オブジェクトまたは固有値

ディクショナリ "ofValues" は､キーとデータをアソシエートオブジェクト内で重ね合うことなく直接格納します。ディクショナリ "ofAssociations" とディクショナリ "ofValues" はそれぞれどのようなときに使うのでしょうか。

ディクショナリ "ofValues" は、単にキー項目に対してデータ項目を格納したい場合、実行時の処理速度とメモリ使用効率という点で若干優れています。しかし、キーとデータの組み合わせを管理するのに、アプリケーションでどこにでもアソシエートを使用する場合は、ディクショナリ "ofAssociations" の方が適しています。これは、実際のディクショナリはアソシエートに対するオブジェクトハンドルを格納するだけで、ディクショナリに入れる前にアソシエートからキーと値を抽出する必要がないからです。

ディクショナリの作成例（キーおよびデータが固有値の場合）

この例では、キーおよびデータが固有値の場合のディクショナリの作成と、テンプレートの作成について示します。

```
00001  local-storage section.

00002  01  aKeyTemplate      object reference.

00003  01  aDataTemplate    object reference.

00004  01  anAssocTemplate   object reference.

00005  01  aDictionary       object reference.

00006  01  aLength           pic x(4) comp-5.


    ...
```

```
00007  procedure division.
       ...
00008      move 3 to aLength
00009      invoke CobolCompX "newClass" using aLength
00010                            returning aKeyTemplate
00011      move 20 to aLength
00012      invoke CobolPicX "newClass" using aLength
00013                           returning aDataTemplate
00014      invoke Association "newClass using aKeyTemplate
00015                                      aDataTemplate
00016                            returning anAssocTemplate
00017      invoke Dictionary "ofValues" using anAssocTemplate
00018                                      aLength
00019                            returning aDictionary
```

行1～6	テンプレートのための記憶領域を宣言します。
行8～9	PIC X(3) COMP-X数値キーのためのテンプレートを作成します。
行11～12	PIC X(20)データ部分のためのテンプレートを作成します。
行14	アソシエートのテンプレートを作成します。
行17	値のディクショナリを作成します。

15.2.1「固有データクラスのクローン化」を参照してください。

14.5 要素の比較

コレクションは、要素について2種類の比較を行います。

- 等価性
- 他の要素より大きな要素

14.5.1 等価性

コレクションは、1つの要素にメッセージ"equal"を送り、別の要素をパラメータとして与えることにより、2つの要素を比較します。Baseにあるデフォルトのメソッド"equal"は、2つのオブジェクトのオブジェクト参照を比較するだけです（同じオブジェクトの場合だけ等価と判断します）。提供されているクラス ライブラリのオブジェクトの中には、2つのオブジェクトのインスタンス データを比較して等価かどうかを判断するような、独自のバージョンを実現しているものもあります。

多くのアプリケーションで、等価性を判断するために要素のインスタンスデータの一部または全体を比較したい場合があります。独自のバージョンの"equal"メソッドを実現し、デフォルトで提供されているBaseを無効にすることで、これを行うことができます。

"equal"メソッドへのインタフェース

"equal"メソッドへのインタフェースは次のように実現します。

```
method-id. "equal".
linkage section
01 lnkObject              object reference.
01 equalityResult         pic x comp-5.
 88 isEqual               value 1.
 88 isNotEqual             value 0.
procedure division using lnkElement returning equalityResult.
*lnkElementを自身と比較しequalityResultを返すコード
   exit method.
end method "equal".
```

14.5.2 オブジェクトの相対値

SortedCollectionインスタンスのデフォルトのソートメソッドは、コレクション内のすべての要素に"lessThanOrEqual"メッセージ（コレクション"ofReferences"の場合）または"lessThanOrEqualByLength Value"メッセージ（コレクション

"ofValues"の場合）を送ります。このためのデフォルトメソッドはBaseにはありませんが、固有クラスとCharacterArrayによって実現されます。

2つのオブジェクトがそれらのインスタンスデータの一部を使ってそれら自身を比較できるようにするためには、このメソッドを実現する必要があります。

"lessThanOrEqual" メソッドへのインタフェース

SortedCollection のインスタンスは、その相対値を他のオブジェクトと比較したいとき、"lessThanOrEqual" メソッドをオブジェクトに送ります。"lessThanOrEqual"メソッドを実現するには、次のインタフェースを使用します。

```
method-id. "lessThanOrEqual".
linkage section.
 01  lnkString              object reference.
 01  lnkResult              pic x comp-5.
     88   isEqual           value 1.
     88   isNotEqual         value 0.

procedure division using lnkString returning lnkResult.
*lnkElementを自身と比較しlnkResultを返すコード
  exit method.
end method "lessThanOrEqual".
```

"lessThanOrEqualbyLengthValue" メソッドへのインタフェース

“ lessThanOrEqualbyLengthValue ”メソッドを実現するには、次のインタフェースを使用します。

```
method-id. "lessThanOrEqualbyLengthValue".
linkage section.
```

```
01  lnkElement              object reference.
01  lnkSize                 pic x(4) comp-5.
01  lnkCompareResult        pic x comp-5.
    88   isLessThanOrEqual   value 1.
    88   isNotLessThanOrEqual        value 0.


procedure division using lnkElement lnkSize
returning lnkCompareResult.
*lsElementを自身と比較しlsElementが自身よりも
*大きければisNotLessThanOrEqualを返すコード。
*lsSizeパラメータはメッセージ"size"への応答にある
*要素によって返される値
  exit method.
end method "lessThanOrEqualbyLengthValue".
```

14.5.3 要素のハッシュ

ディクショナリクラスとセットクラスは、ハッシュ値を使って要素を格納および検索します。このとき、ハッシュ値を取得するためにオブジェクトに "hash" メッセージを送ります。Baseにあるデフォルトの "hash" 機構はオブジェクトにオブジェクト参照を返します。

デフォルトのハッシュ機構を上書きし、オブジェクトのインスタンスデータを使ってハッシュ値を作成するハッシュ機構を使う必要がある場合がよくあります。これには2つの理由があります。

- デフォルトメソッドから返されるハッシュ値は異なるので、ディクショナリとセットは、インスタンスデータが重複する要素またはキーを追加できるようにするため
- データの検索がより効率的なため。デフォルトメソッドでは、要素またはキーを検索するために返されるハッシュ値は、要素またはキーのデータとは関連性がなく、セットまたはディクショナリは正しいオブジェクトが見つかるまで検索する必要があります。

別々のオブジェクトが同じ値を持っている場合、"hash"によって同じ結果を返さなければなりません。ハッシュ値は要素ごとに固有である必要はありませんが、ディクショナリに入っている重複ハッシュ値が多いほど、要素の記憶

域と検索の効率は低くなります。ハッシュ値は常に正の値にする必要があります。

"hash" メソッドへのインタフェース

"hash" メソッドは、オブジェクトのハッシュ値を返す必要があります。 "hash"へのインタフェースは次のように実現します。

```
method-id. "hash".

linkage section.

 01  lnkHashValue           pic s9(9) comp-5.

procedure division returning lnkHashValue.

*オブジェクトの属性に基づいてハッシュ値を返すコード

  exit method.

end method "hash".
```

14.6 表示機構

コレクションの内容を表示するのに2種類の機構があります。コレクションにメッセージ"display"を送ってテキストウィンドウに表示するか、コレクションをListBoxに表示します。

14.6.1 テキスト ウィンドウでのコレクションの表示

コレクションにメッセージ"display"を送ると、コレクションのすべての要素がコンソール上にかっこ"( )"で囲まれて表示されます。このためには、コレクションのすべての要素にメッセージ"display"を送ります。Baseにあるデフォルトのメソッド"display"は、オブジェクトの記述を表示します。例えば、このメッセージをチュートリアルで使用したParrotオブジェクトのいずれかに送ると、デフォルトの"display"メソッドは次の情報を表示します。

```
An instance of Parrot
```

CharacterArrayクラスとCOBOL固有クラスは、それぞれの内容を表示するために、それぞれ独自のメソッド"display"を実現しています。独自のオブジェクトの内容を表示できるように、そのオブジェクト用の"display"を実現することができます。

"display" メソッドへのインタフェース

"display" メソッドを実現するには、次のインタフェースを使用します。

```
method-id. "display".
```

```
procedure division .

*コンソール上に自身を表示するコード

  exit method.

end method "display".
```

14.6.2 リストボックスでのコレクションの表示

"setContents"メッセージを使用して、CharacterArrayまたは固有データのコレクションをListBoxに渡すことによって、それを表示することができます。ListBoxは、コレクションの各要素を1行の文字列として表示します。

リストボックスでのコレクションの表示方法の例

CharacterArray または COBOLPicX オブジェクトのコレクションをリストボックスで表示するには、次のように記述します。

```
invoke aListBox "setContents" using aCollection
```

14.7 コレクションのソートメソッド

SortedCollectionのインスタンスは、要素を追加された順にソートします。デフォルトのソート機構は、要素の相対値を比較するために、要素に"lessThanOrEqual"（コレクション"ofReferences"の場合）または"lessThanOrEqualbyLengthValue"（コレクション"ofValues"の場合）を送ります（14.5.2「オブジェクトの相対値」を参照）。

SortedCollectionのインスタンスのためのソートメソッドを独自に設定することができます。作成したメソッドは、2つの要素を比較し、どちらの要素が先になるかを示すための値を返さなければなりません。

デフォルトのソートメソッドを上書きするには、そのメソッドのためのCallBackを作成し、そのCallBackをSortedCollectionに渡します。

ソートメソッドへのインタフェース

SortedCollectionインスタンスのためのソートメソッドへのインタフェースは次のように実現します。

```
method-id. "sortMethod".

 linkage section

 01 lsElement1            object reference.
```

```
01 lsElement2            object reference.
01 orderResult           pic x comp-5.
 88 element1first         value 1.
 88 element2first         value 0.
procedure division using lsElement1 lsElement2
             returning orderResult.
*lsElement1をlsElement2と比較し、どの要素が他のものに
*先行するかを決定するために
*orderResultを設定するコード
   exit method.
end method "sortMethod".
```

Callback としてコレクションにメソッドを渡すため、いずれのオブジェクトでもそれを実現でき、どんなメソッド名でも付けることができます。例としてここでは"sortMethod" を使用しています。

14.8 コレクションの繰返しメソッド

コレクションクラスには、1つのメッセージをコレクションに送ることによって、コレクションのすべての要素へのアクセスを可能にする、繰返しメソッドが4つあります。このような方法を使用するには、CallBackオブジェクトを与える必要があります。CallBackは、コレクションの各要素にパラメータとして渡されます。

提供されている繰返しメソッドは次のとおりです。この項では、CallBackメソッドがこれらと共に動作するために持っていなければならないインタフェースを示します。

"do"　　コレクション内のすべての要素をCallBackに渡します。

"select"　　コレクション内のすべての要素をCallBackに渡します。CallBackが（PIC X COMP-5を値1に設定することによって）"true"を返した要素は、新しいサブコレクションに格納されます。

"reject"　　コレクション内のすべての要素をCallBackに渡します。CallBackが（PIC X COMP-5を値0に設定することによって）"false"を返した要素は、新しいサブコレクションに格納されます。

"collect"　　コレクション内のすべての要素をCallBackに渡します。CallBackはオブジェクト（もとの要素

と同じ場合も違う場合もある）を返します。返されたオブジェクトは、すべてもとのコレクションと同じ型の新しいコレクションに集められます。

繰返しメソッドを使用するためには、次の手順に従います。

1. コレクションのすべての要素を処理するメソッドを用意します。
2. 作成したメソッドに対するCallBackを作成します。
3. 上にリストした繰返しメソッドの1つへ、コレクションに対するCallBackをパラメータとして渡します。

繰返しの使用方法の例

この例は、コレクション内の要素を処理するために、コレクションの繰返しを使用するステップの概要を示しています。

1. 要素を処理するメソッド用に callback を作成します。

```
invoke Callback "new" using self "processElement "
                returning processElementMethod
```

2. コレクションはすべてのデータ要素を processElementMethod (このオブジェクトの "processElement" メソッド) に渡します。

```
invoke theCollection "do" using processElementMethod
                returning temp
```

3. 実行がこのメソッドから戻るとき、theCollection の各要素は processElement メソッドに渡されます。

"do" メソッドへのインタフェース

"do"のためのCallBackメソッドのインタフェースは、次のように実現します。

```
   method-id. "doMethod".
   linkage section.
   01  lnkElement                    object reference.
   procedure division Using lnkElement.
   *要素を処理するコード
     exit method.
   end method "doMethod".
```

Callback としてコレクションにメソッドを渡すため、いずれのオブジェクトでもそれを実現でき、どんなメソッド名でも付けることができます。例としてここでは"doMethod" を使用しています。

"select" メソッドへのインタフェース

"select"のためのCallBackメソッドのインタフェースは、次のように実現します。

```
    method-id. "selectMethod".
    linkage section
    01 lnkElement             object reference.
    01 selectResult           pic x comp-5.
     88  selectThisElement    value 1.
     88  dontSelectElement    value 0.
    procedure division using lnkElement returning selectResult.
```

***新しいサブコレクションに**`lnkElement`**を含めるかどうかを**

***決定するためのコード**

```
       exit method.
    end method "selectMethod".
```

Callback としてコレクションにメソッドを渡すため、いずれのオブジェクトでもメソッドを実現でき、どんなメソッド名でも付けることができます。例としてここでは"selectMethod" を使用しています。

"reject" メソッドへのインタフェース

"reject"のためのCallBackメソッドのインタフェースは次のように実現します。

```
    method-id. "rejectMethod".
    linkage section
    01 lnkElement            object reference.
    01 rejectResult          pic x comp-5.
```

```
 88 rejectThisElement      value 1.
 88 dontRejectElement      value 0.
procedure division using lnkElement returning rejectResult.
*新しいサブコレクションにlnkElementを含めるかどうかを
*決定するためのコード
   exit method.
end method "rejectMethod".
```

Callback としてコレクションにメソッドを渡すため、いずれのオブジェクトでもメソッドを実現でき、どんなメソッド名でも付けることができます。例としてここでは "rejectMethod" を使用しています。

"collect" メソッドへのインタフェース

"collect"のためのCallBackメソッドのインタフェースは次のように実現します。

```
method-id. "collectMethod".
linkage section
01 lnkElement            object reference.
01 returnElement          object reference.

procedure division using lnkElement returning returnElement.
*新しいコレクション用の要素を戻すコード
   exit method.
end method "collectMethod".
```

Callback としてコレクションにメソッドを渡すため、いずれのオブジェクトでもメソッドを実現でき、どんなメソッド名でも付けることができます。例としてここでは "collectMethod" を使用しています。

第15章 固有データ

Object COBOLには、固有データにメッセージを送るための機構があります。この章では、クラス ライブラリでサポートされている各種の固有データにメッセージを送る方法と、他の型をサポートするためのクラスの作成方法について説明します。

15.1 概要

Object COBOLのクラス ライブラリには、COBOLの固有データ型に対応したクラスが入っています。これらのクラスのオブジェクトは、COBOLのデータ項目に対応しています。これにより、オブジェクト指向の方法によって固有データを格納および操作できます。

固有データ機構は、コレクションクラスが固有データを格納するのに使われます。このとき、コレクション内の各固有データ項目ごとにオブジェクトを作成する必要はありません。固有データは、データがコンパイル時にCompiler(コンパイラ)によって割り当てられる静的オブジェクトと考えることができます。Micro Focus Object COBOLの他のオブジェクトは、すべて動的で、データは実行時に割り当てられます。

15.2 固有データの使い方

固有データクラスは次のようなことに使用できます。

- 固有データをコレクションに格納する (コレクションを作成するためのテンプレートとして、固有データ クラスを使用します)
- 実行時にインスタンスを作成せずに固有データに直接メッセージを送る
- 固有データが入った動的インスタンスを作成する

提供されている固有クラスは、現在の長さのデータを格納できるだけです。他の任意の長さの固有データに使用したい場合、クラスをクローン化し、求める長さのデータについて新しいクラスを作成します。

クラス ライブラリでは次の3種類のCOBOLデータ型がサポートされています。

- PIC X(n)
- PIC X(n) COMP-X
- PIC X(n) COMP-5

他の型の固有データを使用したい場合は、固有データからサブクラスを作成して、固有データ クラスを作成することができます。

15.2.1 固有データクラスのクローン化

固有データを表すために3種類のクラスがあります。それぞれはあらかじめ決められた長さのデータが格納できるインスタンスを作成します。

クラス	データ長
COBOLPICX	1バイト
COBOLCOMP5	4バイト
COBOLCOMPX	4バイト

COBOLPICXは1バイト長のデータを格納できます。COBOLCOMP5とCOBOLCOMPXは4バイト長のデータを格納できます。異なる長さのデータを格納するには、別の長さ用にクラスをクローン化する必要があります。 クラスをクローン化するときは、新しいクラス オブジェクトが、別の長さのデータの格納用にセットアップされるということ以外は、クローン化する元のクラスと同じ属性を持つように作成します。

これには、いずれかの固有データクラスにメッセージ"newClass"を送ります。このとき、パラメータとして長さを指定します。"newClass"メソッドによって与えられた長さのデータを格納できるクラスが返されます。

クローン化したクラスは、INVOKE...ASのためのテンプレートとして、および固有値のコレクションを作成するために使用できます。固有クラスのインスタンスは、"new"メソッドを使って作成することもできます。

固有クラスのクローン化の例

この例では、固有のクラスのクローン化の方法を説明します。これは、特定の長さでデータを格納するように初期化された、新しいクラスとオブジェクトを作成するためのものです。この例では、6バイト長の文字列を扱うためのクラスにクローン化されたcobolPicxを示します。

```
    move 6 to aLength
    invoke cobolPICX "newClass"
                   using aLength
                returning PicX6Class

   aLength      PIC X(4) COMP-5として宣言します。
```

```
PicX6Class   OBJECT REFERENCEとして宣言します。
```

関連項目

- 14.3「コレクションの作成」

15.2.2 固有データ型へのメッセージの送信

INVOKE...ASを使ってCOBOLの固有データにメッセージを送ることができます。 INVOKE...AS 構文により、インスタンス データとして固有データを使用し、呼び出されるメソッドを実行するクラスを指定できるようになります。それによって、静的なオブジェクトとして固有データを取り扱うことが可能になります。 他のオブジェクトと同様に、このメッセージをインスタンスが理解できない場合、継承チェーンのスーパークラスに渡されます。

クラス ライブラリにあるクラスを使って、PIC X、COMP-5、COMP-X として宣言されているデータにメッセージを送信することができます。 まず、自分のデータの長さに、それらをクローン化をすることが必要です。 クラス ライブラリでサポートされていない他の型のデータを使用したい場合は、独自の固有クラスを作成するために、Intrinsicをサブクラス化することができます。

固有データ タイプへのメッセージ送信の例

この例では、固有データ項目に送信されるメッセージを示します。メッセージの送信と同様に、データの宣言を示します。

```
*データの宣言
01 aNumber                pic xx comp-5.
01 comp5Length6           object reference.
...
*メッセージの送信
invoke aNumber as comp5Length2 "hash" reutrning aHashValue
```

上の例では、Comp5Length2 は CobolComp5 クラスのクローンです。クローン化のコードは示していません。クラスをクローン化する方法については、15.2.1「固有データクラスのクローン化」の項を参照してください。

これによって、固有データ項目ANUMBER内に固有データを持った静的オブジェクトにメッセージ"hash"が送られます。他のオブジェクトと同様に、このメッセージをインスタンスが理解できない場合、継承チェーンのスーパークラスに渡されます。

15.3 新しい固有クラスの作成

クラス ライブラリのどのクラスでもサポートしていない固有データ型を使用したい場合、新しい固有クラスを作成します。これには、データと共にIntrinsicから継承されるクラスを作成します。

次の2項では、以下のものを作成するのに必要なコードを取り扱います。

- クラスオブジェクトコード
- インスタンスオブジェクトコード

15.3.1 固有クラスオブジェクトのためのコード

固有クラスのためのクラス初期化コードは、インスタンスによって表されるデータのデフォルトサイズをバイト単位で設定し、それをデータ項目STORAGEREQUIREMENTSに入れなければなりません。これは、次のようにIntrinsicで宣言します。

```
01 storageRequirements   pic x(4) comp-5.
```

クラスのクローン化機構によって、固有クラスのユーザはさまざまな長さのデータを取り扱うことができます。

次のクラスメソッドもコーディングしなければなりません。

- "baseClass"
  このクラスオブジェクトへのオブジェクトハンドルを返します。
- "maximumSize"
  この固有データ型に許される最大バイト数を返します。例えば、COMP-Xデータ項目は8バイト以下なので、COBOLCOMPXクラスの中の"maximumSize"メソッドは8を返します。

"baseClass" メソッドへのインタフェース

"baseClass"のためのメソッドインタフェースは次のようにコーディングします。

```
method-id. "baseClass"


linkage-section.
01   lnkHandle    object reference.


procedure division returning lnkHandle.
```

```
*以下の文中のnameOfThisClass用のクラスの

*クラスIDを置き換える

    set lnkHandle to nameOfThisClass


    exit method.
end method "baseClass"
```

"baseClass"メソッドは、SELFではなく指定されたクラスのオブジェクトハンドルを返します。したがって、固有クラスのクローンにメッセージを送った場合も、baseClassについて正しいハンドルが返されます。

"maximumSize" メソッドへのインタフェース

"maximumSize"のためのメソッドインタフェースは次のようにコーディングします。

```
method-id. "maximumSize".
linkage-section.
01   lnkSize      pic x(4) comp-x.


procedure division returning lnkSize.
*最大割当てサイズを戻すコード
    exit method.
end method "maximumSize".
```

15.3.2 固有インスタンスオブジェクトのためのコード

Collectionクラスで使用するために、固有クラスのインスタンスのための比較メソッドをコーディングしなければなりません。それぞれの比較方法について、2種類のメソッドがあります。1つは固有オブジェクトを別のオブジェクトと比較し、もう1つは固有オブジェクトを値と比較します。

固有オブジェクトを比較するには、次のメソッドを準備する必要があります。

- "equal"
- "lessThan"
- "greaterThan"

これらのすべてのメソッドは、"equal"メソッドと同様のメソッド インタフェースを持ちます。

オブジェクトを固有データ項目と比較するためのメソッドには、次のものがあります。

- "equalByLengthValue"
- "lessThanByLengthValue"
- "greaterThanByLengthValue"

これらのすべてのメソッドは、"equalByLengthValue"メソッドと同様のインタフェースを持ちます。

固有インスタンスの値は、継承されたインスタンスデータ項目INSTANCEDATAに保存されます。これは次のように宣言します。

```
01 instanceData  pic x.
```

1バイト長で宣言していますが、固有データにとって常に正しい長さのメモリ領域を参照します。中のデータを得るには部分参照を使用します。例えば、インスタンスデータの最初の4バイトを見るには、次のように参照します。

```
instanceData(1:4)
```

オブジェクト比較メソッドへのメソッドインタフェース

オブジェクト比較メソッドへのメソッドインタフェースは、次のようにコーディングします。 ここでは、例として"equal"メソッドへのインタフェースを示します。

```
method-id. "equal".


linkage section.
  01 lnkBoolean            pic x comp-x.
     03 isTrue             value 1.
     03 isFalse            value 0.
  01 lnkIntrinsic          object reference.
```

```
procedure division using lnkIntrinsic
                  returning lnkBoolean.
*lnkIntrinsic中の値をこのインスタンス中の値
*と比較するコード。比較の結果が真であれば
*isTrueを設定し、そうでない場合はisFalseを
*設定する
  exit method.
end method "equal".
```

固有データ比較メソッドへのメソッドインタフェース

固有データ比較メソッドへのメソッドインタフェースは、次のようにコーディングします。 ここでは、例として"equalbyLengthValue"メソッドへのインタフェースを示します。

```
method-id. "equalByLengthValue".

linkage section.
  01 lnkBoolean            pic x comp-x.
     03 isTrue             value 1.
     03 isFalse            value 0.
  01 lnkLength             pic x(4) comp-x.
  01 lnkValue              pic x occurs 1 to maxSize.
procedure division using lnkLength lnkValue.
                  returning lnkBoolean
*lnkValue中の値をこのインスタンス中の値と比較
```

```
*するコード。比較の結果が真であればisTrueを設
*定し、そうでない場合はisFalseを設定する。
  exit method.
end method "equalByLengthValue".
```

第16章 CallBackのフレームワーク

Collectionクラス、GUIクラス、ExceptionManagerを使用するには、CallBackを理解する必要があります。この章では、CallBackインスタンスを作成する方法と、それを自分のアプリケーションの中で使用する方法について説明します。

16.1 概要

CallBackクラスのインスタンスには、オブジェクトとメソッドの名前が入っています。CallBackにメッセージ"invoke"を送ると、そこに入っているオブジェクトにメッセージが送られ、指定されたメソッドが起動します。

CallBackインスタンスは、コードのブロックへのポインタのようなものです。CallBackによって、コードブロックをクラスまたはオブジェクトメソッドに送り、後で実行させることができます。例えば、例外取り扱いメソッドを実現するには、そのメソッドが入ったCallBackを作成し、それを例外ハンドラ（ExceptionHandler）に送ります。

16.2 CallBackの使い方

CallBackは2段階で使用します。

- CallBackインスタンスの作成
- CallBackインスタンスの起動

クラス ライブラリのオブジェクトに対してCallBackを使用する場合、CallBackインスタンスをどのように作成するのかを知るだけで十分です。インスタンスはクラス ライブラリオブジェクトに渡され、必要に応じて呼び出されます。しかし、自分で作ったクラスの中でCallBackを使用したい場合もあるので、この章ではCallBackの起動方法についても説明します。

クラス ライブラリには、いずれのCOBOLエントリ ポイントでもCallbackを作成することができる、EntryCallbackと呼ばれるクラスもあります。Callbackを使用できるところであればどこでも、クラス ライブラリのオブジェクトと共に EntryCallback を使用することができます。これにより、クラス ライブラリを使って、手続き上の COBOL コードを使ったコレクションの繰返しのような機能をさせることができます。Callback を使用する場合の規則はすべて、 EntryCallbackにも同様に適用されます。

16.2.1 CallBackの作成

CallBackオブジェクトを作成するには、CallBackクラスにメッセージ"new"を送ります。CallBack内に作成したいメソッドを実現するためのオブジェクトと、そのメソッドを起動するためのメッセージ名を与えなければなりません。Callback の作成時には、オプション パラメータも最大6個まで指定することができます。これらは Callback と共に格納され、呼び出されるたびにCallbackメソッドに渡されます。

Callbackの作成例

CallBackは、次のいずれかの方法で作成します。

```
move "aMethod " to messageName
invoke Callback "new" using anObject messageName
                        [p1] [p2] [p3] [p4] [p5] [p6]
                returning aCallback
```

または

```
invoke Callback "new" using anObject "aMethod "
                        [p1] [p2] [p3] [p4] [p5] [p6]
                returning aCallback
```

メッセージ名のための定数の終わりには1つ以上のスペースを入れます。CallBackのメソッド"new"は、メッセージ名の終わりを見つけるためにこのスペースを探します。

パラメータの説明

anObject	CallBackにより起動されるメソッドが入ったクラスまたはオブジェクトへのオブジェクトハンドルが入っています。
messageName	PIC X(n)データ項目。メソッドのメッセージ名と終了を示すスペースを入れるのに十分な長さで宣言します。
aCallBack	作成するCallBackに対するオブジェクト参照。
[p1]...[p6]	メソッドに関する最大6個のオプションパラメータ。これらは、メソッドを呼び出すたびに送られます。すべてのパラメータはusage object referenceとして宣言しなければなりません。

16.2.2 CallBackの起動

クラス ライブラリ内のオブジェクトに対してCallBackを使用する場合、CallBackの起動方法を知っている必要はあり

ません。しかし、自分で作ったクラスで使用したい場合もあります。CallBackに格納されたメソッドを実行するには、CallBackインスタンスにメッセージ"invoke"を送ります。このメッセージと共に最大3つのパラメータを送ることができます。これらのパラメータは、CallBackの作成時に指定されたパラメータに続けてCallBackメソッドに渡されます（16.2.1「CallBackの作成」の項を参照）。

Callbackの起動例

この例では、Callback の起動方法を示します。

```
invoke aCallback "invoke" [using p1 [p2] [p3] ]
                    [returning result]
```

パラメータの説明

[p1]...[p3]	メソッドの最大6個のオプション パラメータ。これらは、Callback が作成されたとき指定されたパラメータに続いて、Callback内のメソッドへ渡されます。すべてのパラメータはusage object referenceとして宣言しなければなりません。

CallBack内のメソッドがパラメータを返す場合、CallBackの起動時に正しいサイズのreturning句を指定しなければなりません。

第17章 例外取り扱いのフレームワーク

例外取り扱いは、オブジェクトがエラーを発生した時にトラップするための機構です。この章では、この機構を自分で作ったオブジェクトの中で使用する方法について説明します。

17.1 概要

例外取り扱いは、Object COBOLプログラムの中でエラーを柔軟に取り扱うための機構を提供します。オブジェクトがエラーをトラップすると、例外を発生します。エラーの種類は例外番号によって表されます。例外が例外ハンドラによってトラップされない場合、Object COBOLのランタイムコードは例外番号とエラーメッセージを画面に表示します。

付属のクラス ライブラリ内のオブジェクトは、例外を発生することによってエラーに応答します。クラス ライブラリには、1から始まる予約済みの例外番号とエラーメッセージのファイルがあります。アプリケーション内でオブジェクトのための例外条件を定義し、独自のメッセージファイルを対応づけることができます。この場合、例外番号は10001から始めます。例外番号はPIC X(4) COMP-5データ項目によって表され、最大値は4294967295です。

17.2 例外取り扱いの使い方

例外取り扱いは2つの独立した手順からなります。

- オブジェクトに関するエラー条件がどのように構成されているかを判断し、エラーが起こったときに例外を発生するためのコードを作成します。
- このオブジェクトを例外ハンドラ付きで登録するかを決め、適切な例外ハンドラをコーディングします。

アプリケーション内のオブジェクト用に例外ハンドラを作成する必要はありません。例外がトラップされずに、アプリケーションが終了するだけです。例外ハンドラは、"raiseException"機構を使ってオブジェクトによって発生したエラーをトラップするだけで、ランタイムシステムのエラーはトラップしません。

17.2.1 例外の発生

メソッドがエラー（例えば、範囲外の値）を検出したら、例外を発生する必要があります。エラーは、エラー番号（10,000以下の番号はクラス ライブラリによる使用のために予約済み）を割り当てることによって表します。それは、特定のファイル内のエラーメッセージに対応しています。 ExceptionHandlerで登録された各エラー ファイルには、実行時に、数値的なオフセットが割り当てられます。 例外ハンドラに対してエラーを一意に規定する例外ＩＤを取得するために、例外を発生する前に、エラー番号にエラー ファイルのオフセットを加算します。

例外を発生するオブジェクトは、エラー　メッセージ　ファイルが登録されてるオブジェクトでないこともあります。したがって、対応するエラー　ファイルのためのオフセットを取得するために、ExceptionManagerを照会するこ

とができます。 次に、オブジェクトは、"raiseException"メッセージをエラー番号と共に自分自身に送ることによって、例外を発生させます。

```
invoke self "RaiseException" using errorNumber
```

"RaiseException"メソッドは、Baseクラスに実現されており、すべてのサブクラスによって継承されます。オブジェクトに関して登録された例外ハンドラがない場合、システムの例外ハンドラがエラー番号とメッセージを表示し、アプリケーションを終了します。

オブジェクトに関して例外ハンドラを登録した場合、例外ハンドラが起動した後、実行点は"raiseException"の次の文に戻ります。

ExceptionHandlerは、優先度の高い順に登録された例外ハンドラを呼び出します。

1. オブジェクトインスタンス
2. オブジェクトのクラス
3. システム例外ハンドラ

例外を発生する方法

1. ExceptionManager に"queryMessageFile"を送信して、この型のオブジェクトのエラー オフセットを取得します。
2. エラー番号にエラー オフセットを加算して、例外 ID を計算します。
3. パラメータとして例外 ID を使用し、自身に"raiseException"メッセージを送信します。

例外の発生例

この例では、例外の発生方法を示します。エラー オフセットを取得するには、ライブラリ ファイルおよびエラー ファイルの名前を渡します。そのエラー ファイルは、そのライブラリ ファイル内になければなりません。また、ExceptionManager を使って事前に登録されている必要があります。

ErrorOffset, ErrorNumberおよびExceptionIdは、すべてPIC X(4) COMP-5として宣言されています。

- エラー オフセットを取得します。

```
invoke ExceptionManager "queryMessageFile" using "mflang"
                                                  "err-file"
                                                returning errorOffset
```

- 例外 ID を計算します。

```
add errorOffset to errorNumber giving exceptionId
```

- 例外を発生します。

```
invoke self "raiseException" using exceptionId
                        returning anObject
```

"raiseException" でパラメータを返すことにより、例外ハンドラは、例外を発生するオブジェクトに情報を戻すことができます。

17.2.2 例外ハンドラによるオブジェクトの登録

ExceptionManager を使って、任意のクラスまたはインスタンス オブジェクトに対して例外ハンドラを登録することができます。 例外ハンドラは、例外の処理方法を知っているメソッドに対するCallbackです。 オブジェクトが例外を発生すると、登録されたエラー取り扱いメソッドが起動します。

クラス オブジェクトに対する例外ハンドラを登録すると、そのクラスのすべてのインスタンスもまた同じ例外ハンドラに対して自動的に登録されます。 特定のインスタンスに対して例外ハンドラを登録することで、その登録は上書きすることができます。

例外ハンドラの登録例

オブジェクトを使って例外を登録するには、次のようにします。

```
invoke ExceptionManager "register" using anObject
                                    anExceptionMethod
```

anObjectは、登録したいオブジェクトへの参照(オブジェクト)、anExceptionMethodは例外メソッドへのCallBackです。CallBackについては、第16章「CallBackのフレームワーク」を参照してください。

17.2.3 例外ハンドラ登録のキャンセル

オブジェクトに対して登録されている例外ハンドラは、いつでもキャンセルすることができます。 これを行うと、オブジェクトによって発生される例外は、システムの例外ハンドラによって取り扱われます。 例外ハンドラのオブジェクトの登録をキャンセルするには、Exceptionmanager に"cancel"メッセージを送ります。

例外ハンドラのキャンセル例

この例では、クラスまたはインスタンス オブジェクトに対して登録された例外ハンドラをキャンセルする方法を示します。

```
invoke ExceptionHandler "cancel" using anObject returning aBool
```

anObjectは登録解除するオブジェクト、aBoolはPIC X COMP-5です。登録解除が成功した場合、ここに値1が入ります。

17.2.4 例外メソッドの作成

例外メソッド（exception Method）を使うと、Object COBOLのオブジェクトによって発生した例外をアプリケーションにとって最適な方法で取り扱うことができます。 クラスまたはインスタンス オブジェクトに対して例外メソッドを登録することにより、それらのオブジェクトが例外を発生するときにはいつでも例外メソッドが呼び出されるようにできます。

例外メソッドには、例外を発生したオブジェクトに対するハンドルと、例外 IDが渡されます。 例外 ID からエラー番号を取得するには、まずExceptionManagerから、オブジェクトで発生したタイプのエラーに対するエラー オフセットを取得し、例外ID からそのオフセットを減算する必要があります。

例外メソッドへのインタフェース

この例では、例外メソッドへのインタフェースをコーディングする方法を説明します。また、例外 ID からオブジェクトのエラー番号を取得する方法についても説明します。例外ハンドラは、例外を発生したオブジェクトのエラー ファイル名を知る必要があります。例外ハンドラが様々な種類のオブジェクトからのエラーを取り扱うことが予想される場合は、例外を発生したオブジェクトを照会するメソッドを作成する必要があります。

```
    method-id. "exceptionMethod".

    local-storage section.

    01  errorNumber           pic x(4) comp-5.


    01  lsOffset              pic x(4) comp-5.


    linkage section

    01  anObject              object reference.

    01  anExceptionId         pic x(4) comp-5.
```

```
 01  aDataItem              object reference.


procedure division using anObject anExceptionId
              returning aDataItem.
```

*** エラー番号を計算する**

```
 invoke ExceptionManager "queryMessageFile"
                         using "err-lib" "err-file"
                       returning lsOffset

 subtract lsOffset from exceptionId giving errorNumber
```

- エラーを処理します。

```
 evaluate errorNumber
     when knownError1
```

- エラーを取り扱います。

```
     when knownError2
```

- エラーを取り扱います。

```
 when other
```

- 不明なエラーのため、例外を発生します。

```
     invoke self "raiseException" using anExceptionId
 end-evaluate
     exit method.
```

```
    end method "exceptionMethod".
```

パラメータの説明

anObject	最初にエラーを発生したオブジェクト
anExceptionId	発生したエラーを表す番号
aDataItem	例外メソッドによって返されるオブジェクト。これによって、例外を発生したメソッドからデフォルトパラメータを返すことができます。この場合、"raiseException"メソッドから受け取ったパラメータを返すためのコードを、メソッドに与える必要があります。

例えば、HighRateAccountクラスの"openAccount"メソッド（¥demo¥oopsディレクトリのhaccount.cblにある）は、口座開設時の残高が一定額以下の場合に例外を発生します。"openAccount"メソッドからの結果として"raiseException"メソッドから返されたパラメータを返します。これにより、例外ハンドラがある場合、ヌル値によってアプリケーションの残りの部分に例外が起こったことを知らせることができます。詳しくは、8.3「例外ハンドラの作成」の項を参照してください。

例外メソッドはエラー番号を取得すると、期待した値であるかどうかを調べるためにテストします。 期待する値でない場合、再び例外を発生します。この場合、そのエラーの処理方法を知っている例外ハンドラがエラーをトラップするか、システム例外ハンドラがアプリケーションを終了します。

例外ハンドラに対して例外メソッドを登録する前にCallBackを作成しなければなりません（第16章「CallBackのフレームワーク」を参照）。

17.2.5 システム例外メソッド

システム例外メソッドは、システム全体について登録された例外メソッドです。このメソッドは、例外ハンドラに対して明示的に登録されていないオブジェクトが例外を発生したときに起動します。デフォルト動作では、エラー番号をエラーメッセージと共に表示し、プログラムを終了します。

システム例外メソッドを独自のメソッドに置き換えることができます。これには、例外メソッドのためのCallBackを作成し、ExceptionManagerにメッセージ"setSystemHandler"を送ります。

"setSystemHandler"メソッドは、既存のシステム例外ハンドラ用のCallbackに対するハンドルを返します。 このハンドルは格納しておく必要があります。それによって、自分で置き換えたシステム例外メソッドが処理方法を知らない例外を取得した場合に、いつでも、以前のシステム例外メソッドへのCallbackを呼び出してパラメータとして例外ID を渡し、例外を元のメソッドに渡すことができます。

システム例外メソッド置き換えの例

この例では、システム例外メソッドを自分自身で作成したものと置き換えます。

```
invoke ExceptionManager "setSystemHandler" using newHandler
                                    returning oldHandler
```

パラメータの説明

newHandler	例外メソッドのためのCallBackに対するオブジェクトハンドル
oldHandler	置き換えたシステム例外メソッドのためのCallBackに対するオブジェクトハンドル

17.3 エラー メッセージ ファイルの作成

エディタを使用してエラー メッセージ ファイル(ファイル名に拡張子 .err を持つファイル)を作成し、それを.lngファイルにコンパイルします。 その後で、.lngファイルを.lbr ファイルに入れ、.lng および .lbr ファイルをExceptionManagerを使って登録します（詳細については、次の「エラー メッセージ ファイルの作成方法」および「エラー メッセージ ファイルの例」を参照してください）。

.err ファイルを作成するときは、次の文で始めます。

```
$quote x
$set 1
```

ただし、*x* は引用符として使用したい文字です。

その後、次の形式で各メッセージを作成します。

```
nnnnn  "your error message here"
```

各メッセージには番号を付け、メッセージテキストは引用符で囲まなければなりません。メッセージテキストを始める前に$QUOTEコマンドによって引用符文字を設定します。.err ファイルでは任意の位置で引用符文字を変更することができます。その変更は、次の $quote 文までのすべてのメッセージに適用されます。 また、最初のメッセージの前に$SETコマンドを使用します。注釈を入れるには、$とスペースで行を始めます。

エラー メッセージ ファイルの作成方法

1. エディタを使って、拡張子 .err を持つファイル名のファイルを作成します。
2. 各メッセージごとに番号を付けた項目を作成します。
   その番号は、例外の結果としてそのメッセージを表示するときに使われる、エラー番号になります。
3. .err ファイルをアプリケーションのプロジェクトに追加します。

「COBOL 開発環境」で、これを .lng ファイルにコンパイルします。

4. .lng ファイルを右クリックして、Micro Focus ライブラリ（.lbr）ファイルとしてこれをパッケージ化します。

ライブラリ ファイルには、*xxxxxx*df.lbr という名前を付けます。例えば、ソースコードで mylib として参照されるエラー ライブラリは、mylibdf.lbr というファイル名を持ちます。

5. ExceptionManagerを使って.lbr および .lng ファイルを登録し、"raiseException" メッセージを送ってエラーメッセージを表示します。

ライブラリ ファイルが含まれるディレクトリを指す環境変数 COBDIR を設定して、ExceptionManagerがライブラリファイルを検索できるようにする必要があります。

エラー メッセージ ファイルの例

この例では、エラー メッセージ ファイルを示します。ドル記号 ($) とスペースで行を始めることにより、注釈を入れることができます。

```
$quote "
$
$set 1
1.   "Invalid key type"
2.   "Failed to open file for dictionary"
3.   "Using a key of different size"
4.   "Record exceeds maximum allowed length"
5.   "Error reading control record"
6.   "Indexed file has wrong key structure"
7.   "Key not found in file"
```

```
8.   "Attempt to write a record which has been deleted"


$set 1

          ...


$ ****************************************************************
$ Copyright (c) 1981,1995 Micro Focus Ltd
$
$ Last modified on 95/08/01
$
$ ****************************************************************
```

17.4 エラー メッセージ ファイルの登録

オブジェクトが例外を発生する前に、ExceptionManagerを使ってエラー ファイルが登録されていなければなりません。 ExceptionManagerを使ってエラー ファイルを登録すると、エラー　ファイルを一意に識別するオフセット番号が返されます。

任意の数のクラスまたはオブジェクトが同じエラー ファイルを共用できます。また、同じエラー ファイルが複数回、登録されていても関係ありません。ExceptionManagerはアプリケーションの特定の実行中に常に同じオフセットを返します。

エラー ファイルの登録例

この例では、ExceptionManager を使って登録されているエラー ファイルを示します。最初のパラメータはエラー ファイルが入っているライブラリの名前で、2番目のパラメータはエラー ファイルの名前です。戻される値は、例外が発生したときに一意の例外 ID を作成するためのオフセットです。

```
move spaces to libraryFile, errorFile

move "mflang" to libraryFile

move "err-file" to errorFile

invoke ExceptionManager "registerMessageFile"
```

```
    using libraryFile errorFile

returning anOffset
```

データ項目の宣言

---

errorFile pic x(8)

libraryFile pic x(6)

anOffset pic x(4) comp-5.

第18章 GUIクラスのフレームワーク

この章では、Object COBOLのGUIクラスによって使用されるフレームワークについて説明します。この章の内容は、サブクラス化によってウィンドウやダイアログボックスを作成したり、インタフェースを実行するためのアプリケーションコードを作成するのに役立ちます。

18.1 概要

GUI クラスのフレームワークは、アプリケーションのグラフィカル インタフェースの作成に関連しています。 グラフィカル インタフェースの作成には、以下の2つの部分があります。

- エンドユーザが対話するデスクトップ上のビジュアル オブジェクトの作成
- インタフェースからのイベントの処理

簡単に作れるようにするために、GUI アプリケーションのテンプレートが提供されています。これは自分のアプリケーションを構築するときの土台として使用することができます。 テンプレートはアプリケーションのスケルトン(骨組み)であり、コンパイルし、実行すると、メニュー バー、ツールバー、ステータス バー、さらにアプリケーションの情報を表示できる「About」ボックスを持つリサイズ可能なウィンドウを表示します。

18.2 イベント取り扱い

GUIクラスのフレームワークは、イベントの取り扱いとGUIオブジェクト上へのオブジェクトの表示に関係します。イベントは、ユーザと、インタフェース、アプリケーションコードの間のリンクです。

例えば、ユーザがマウスを動かしたり、ボタンをクリックしたり、キーを押した場合、オペレーティングシステムはアプリケーションにイベントを通知します。WindowsのAPIを使ってインタフェースをコーディングする場合、これらのイベントをすべてポーリングし、関心のあるイベントに対して応答する必要があります。

GUI クラス ライブラリは、関心のあるイベントを指定できるようにすることによって、インタフェースの作成を簡単にします。これによって、指定したイベントだけが通知されます。すべてのポーリングはEventManagerクラスによって取り扱われます。このアプリケーションがアイドル状態の場合、コンピュータは他のアプリケーションの実行を続けることができます。

イベントに対するポーリングをアプリケーションに開始させたい場合は、EventManager に"run"メッセージを送信します。

デスクトップに表示されるGUIオブジェクトは、オペレーティングシステムからさまざまなイベントを受け取ります。しかし、アプリケーションには登録したイベントだけが通知されます。必要なイベントを登録するには、次の2つの方法があります。

- イベントを他のイベントにマッピングする
- 特定のイベントが起こったときにメソッドが起動するようにする

起こり得る各イベントは、イベントIDという固有の識別子を持っています。

18.2.1 イベントIDのフォーマット

GUIオブジェクトが受け取りまたは発生できるイベントは、次のどちらかに分類できます。

- 物理イベント
  マウスまたはキーボードの操作に対する応答としてオペレーティングシステムによって発生します。例えば、ボタンのクリック、メニュー項目の選択、ウィンドウのサイズ変更などがあります。
- 論理イベント
  オブジェクトによって発生します。各アプリケーションには、それぞれ任意の論理イベントのセットを定義できます。 たとえば、GUI アプリケーション テンプレートでは、各ダイアログ ボックスに対して dialog-ok および dialog-cancel 論理イベントが定義されています。

すべてのイベントはPIC X(4) COMP-5データ項目として表されます。論理イベントには、物理イベントにすでに割り当てられたID以外の任意の数値IDを与えることができます。物理イベントのIDはコピー ファイルp2cevent.cpyに定義されています。製品インストール時にデフォルトのディレクトリを選んだ場合、このファイルは¥cobol¥source¥oopsに入っています。イベントの番号は、最低番号からは振らないことをお勧めします。将来リリースされるObject COBOLのGUIクラスは、現在よりはるかに多くの物理イベントを取り扱えるようになる予定です。

Object COBOLに付属のGUIによって認識されるすべての物理イベントには、コピー ファイルp2cevent.cpyの中でレベル78データ項目にIDが割り当てられています。

18.2.2 イベントからのメソッドの起動

イベントが起こったときにメソッドが起動するように、イベントに対する関係を登録することができます。これには、GUIオブジェクトにメッセージ"setEventTo"を送ります。 "setEventTo"メッセージを使用する代りとして、ボキャブラリ機能を使用して、新しい動詞句 MAP EVENT を定義しました。これにより、同じ事を、より自然な言語を使って行うことができるようになりました。 MAP EVENT 動詞の詳細については、18.3「GUIイベント」を参照してください。

例えば、ウィンドウにプッシュボタンがある場合、プッシュボタンが押されるたびにウィンドウにメッセージを送りたいことがあります。 種々のタイプのグラフィカル オブジェクトで利用可能なイベントの詳細については、18.3「GUIイベント」を参照してください。

このクラス ライブラリに付属のすべてのGUIオブジェクトはメッセージ"setEventTo"に応答します。"setEvent"というメソッドもあります。このメソッドは"setEventTo"と同じ機能ですが、ターゲットオブジェクトとメソッド名を提供するかわりに、CallBackをパラメータとして渡します。

メソッドへのイベントのマッピング例

この例のコードでは、プッシュボタンが押されるたびにaWindowがメッセージ"buttonPushed"を受け取るようにaButtonをセットアップします。

```
map event p2ce-clicked upon aButton

        to method z"buttonPushed" upon aWindow
```

この例では、P2CE-CLICKEDがコピー ファイルp2cevent.cpyの中で宣言された物理イベントIDの1つです。 このコピー ファイルは GUI クラス ライブラリの一部として提供されます。

注記：

この例では NetExpress のボキャブラリ機能を使って定義されているMAP EVENT動詞を使用しています。この動詞を使用するには、プログラムの先頭の、CLASS-ID または PROGRAM-ID 見出しの前に次の文を入れる必要があります。

```
copy "guibase.if ".
```

18.2.3 イベントの再マッピング

物理イベントを論理イベントにマップできます。これは、アプリケーションオブジェクトをユーザインタフェースと切り離すのに役立ちます。例えば、付属のBankアプリケーションのデモンストレーションでは、「Add」ボタンを押すか、メインウィンドウの「File」メニューから「Add」を選択すると、Bankウィンドウで論理イベントadd-recordが発生します。

制御するアプリケーションオブジェクトは、Bankウィンドウ上のadd-recordイベントへの関係を登録しています。その結果、ユーザがレコードを追加したい場合、アプリケーションにそのことが通知されます。アプリケーションは、Bankウィンドウのコントロールについて知っている必要はなく、ユーザがコントロールを操作したときに何を要求しているのかがわかるだけでかまいません。

論理イベントへの物理イベントのマッピング例

この例では、論理イベントに物理イベントをマップする方法を示します。コードの最初の部分では、プッシュボタン上の物理イベントを論理イベントにマップします。この場合、論理イベントはプッシュボタンの親オブジェクト(通常、ダイアログ ボックス)上で発生します。

マッピングをセットアップするコードは、ダイアログ ボックスの初期化コードの一部です。リソースをロードするダイアログ ボックスの場合は、"create" メソッドにそれを挿入します。動的に作成するダイアログ ボックスの場合は、"initialize" メソッドにそれを挿入します。この例では、"create" メソッドでダイアログ ボックスが示されます。

次の2つのコードは、このような物理イベントと論理イベントの間のマッピングをセットアップする方法を示しています。プッシュボタンを所有するウィンドウの初期化コードは、次のようにします。

```
method-id. "create".


local-storage section.
01 lsPhysicalEvent      pic x(4) comp-5.
01 lsLogicalEvent       pic x(4) comp-5.
...
procedure division.
...
```

- ガジェットとコントロールのためのイベント マッピングをセットアップします。 p2ce-clicked は、提供されたコピー ファイル p2cevent.cpy で定義されています。 add-record は、アプリケーション プログラマが定義します。その値は、10,000より大きな数値である必要があります。

```
    map event p2ce-clicked upon aButton
            to logicalEvent add-record upon self
    ...
    exit method.
end method "create".
```

次のコードでは、アプリケーションがADD-RECORDイベントへの関係を登録する方法を示します。ダイアログ ボックスのハンドルは lsDialogBox にあります。

```
map event p2ce-clicked upon aButton
        to method z"addRecord" upon aWindow
```

論理イベントをメソッドにマッピングすることは、物理イベントをメソッドにマッピングするのとまったく同じです。イベント登録機構では、論理イベントと物理イベントを区別しません。

アプリケーションのエンドユーザがウィンドウ上のボタンを押すと、そのボタンはダイアログ ボックス上でADD-RECORDイベントを発生します。アプリケーションオブジェクトには、ADD-RECORDイベントへの関係が登録され

ているので、メッセージ"addRecord"を受け取ります（この部分の機構は、18.2.2「イベントからのメソッドの起動」項の説明と同じです）。

注記：

この例では NetExpress のボキャブラリ機能を使って定義されているMAP EVENT動詞を使用しています。この動詞を使用するには、プログラムの先頭の、CLASS-ID または PROGRAM-ID 見出しの前に次の文を入れる必要があります。

```
copy "guibase.if ".
```

関連項目

- 18.3「GUI イベント」

18.2.4 イベントによって起動されるメソッド用のインタフェース

この項では、イベントによって起動されるメソッド用のインタフェースについて説明します。メソッドは、"setEventTo"メソッドまたは"setEvent"メソッドによって登録されたイベントが起こったときに起動します。

イベントメソッドは、イベントに関する詳しい情報を問い合わせることができるイベントオブジェクトを受け取ります。しかしほとんどの場合、この必要はありません。 イベント オブジェクトは必ず、イベント クラスのサブクラスの 1 つのインスタンスです。 これらのオブジェクトはすべて、"getId"メッセージに応答します。 このメッセージにより、イベントの ID を見つけることができます。 また、これらのオブジェクトは、"getObject"メッセージにも応答します。 このメッセージにより、イベントを起動したオブジェクトのハンドルが取得できます。 各タイプのイベントで戻されるイベント オブジェクトのタイプの詳細については、18.3「GUIイベント」を参照してください。

イベント処理ループから戻った後にイベント オブジェクトを参照する必要がある場合は、そのオブジェクトを最初にコピーする必要があります。 これは、イベント処理機構が、効率を良くするために同一のオブジェクトを再使用するためです。

イベントオブジェクトからイベントを受け取ったオブジェクトのオブジェクトハンドルを、イベントが起こったときのキーボードの状態やマウスの位置と同様に得ることができます。詳しくは、オンラインの『クラス ライブラリ リファレンス』のイベントクラスに関するインタフェースの項を参照してください。

イベントで起動されたメソッドへのインタフェース

イベントによって起動されるメソッドへのインタフェースは次のように実現します。

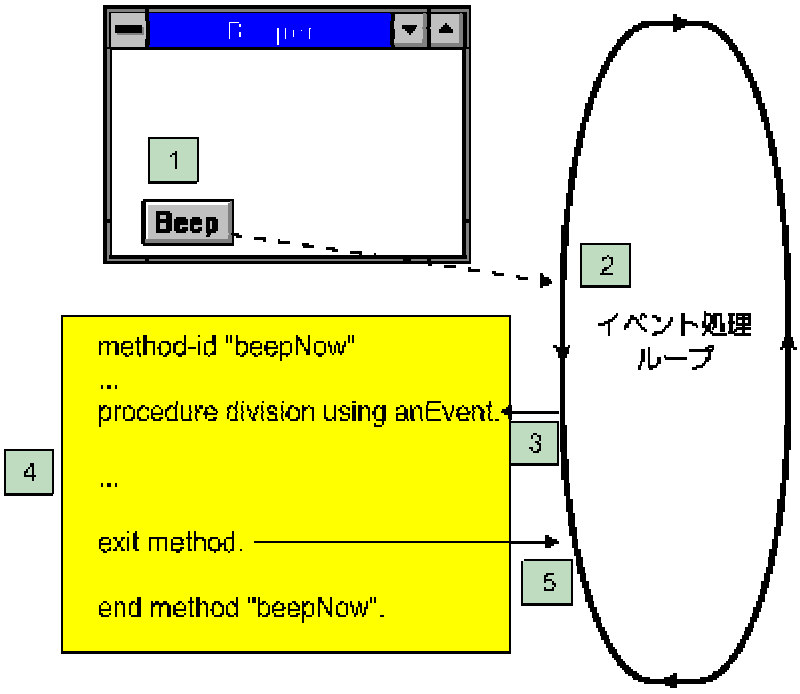
```
method-id. "methodName".
```

```
linkage section.

01  lnkEventObject         object reference.

procedure division using lnkEventObject.

*lnkEventObjectを問合わせてイベントを処理するコード

   exit method.

end method "methodName".
```

18.2.5 EventManager

EventManagerには、イベント処理ループを処理する責任があります。EventManager は、GUI インタフェースのグローバル属性を設定する場所でもあります。 ほとんどのグローバル属性は、他のバージョンの Object COBOL GUI クラス ライブラリとの下位互換のためのものです。

グラフィカル オブジェクトを表示し、アプリケーションをそのオブジェクトからのイベントに応答するようセットアップしたら、イベント処理ループを開始する必要があります。 イベント処理ループが開始されると、アプリケーションはイベントを待ちます。 メソッドにマッピングしたイベントをグラフィカル オブジェクトが受信するたびに、そのメソッドが起動されます。 メソッドが実行されると、アプリケーションはイベント処理ループに戻ります。この流れを、次の図に示します。



1. エンドユーザがボタンをクリックして、イベントが発生します。
2. イベント処理ループが、イベントを見つけます。
3. このイベントへの関係がアプリケーションに登録されているので、メソッドが呼び出されます。
4. エンドユーザによるボタンのクリックを処理するコードが実行されている間は、コントロールはアプリケーション内にあります。
5. exitメソッド文で、コントロールはイベント処理ループに返されます。

18.3 GUI イベント

種々のタイプの GUI オブジェクトは、別のタイプのイベントを受け取ることができます。GUI クラス ライブラリによりサポートされているすべてのイベントはp2cevent.cpyにリストされており、¥netexpress¥base¥source¥guiclディレクトリにあります。

以下に、それぞれのタイプのオブジェクトのイベントをリストします。このリストでは、WindowはWindowのサブクラスを意味し、GadgetはGadget(ガジェット)のサブクラスを意味します(以下同様)。クラス ライブラリの階層を見ると、特定のクラスのサブクラスがどれであるかを調べることができます（以下の各項を参照してください）。

- 18.3.1 Windowイベント
- 18.3.2 Gadgetイベント
- 18.3.3 List Box、Selection Box 、TreeViewイベント
- 18.3.4 Scrollbarイベント
- 18.3.5 UpDownイベント
- 18.3.6 Buttonイベント
- 18.3.7 Mouseイベント

18.3.1 Windowイベント

以下の表は、WindowまたはWindowのサブクラスであるオブジェクトに送信されるすべてのイベントを示したものです。イベントは数値ですが、この表では、¥netexpress¥base¥source¥guiclディレクトリにあるp2cevent.cpyの中で定義されているシンボリック識別子で示します。「イベント オブジェクト」欄は、イベントにより呼び出されるメソッドに渡されるイベントのタイプを示します。イベントのタイプに関する詳細な情報について、イベント オブジェクトを問合わせることができます。

イベント	説明	イベント オブジェクト
p2ce-move	ウィンドウが移動した	MoveEvent
p2ce-resize	ウィンドウがサイズ変更された	SizeEvent
p2ce-close	ウィンドウが閉じた	Event
p2ce-activate	ウィンドウがフォアグラウンド ウィンドウになった	Event
p2ce-deactive	ウィンドウがフォアグラウンド ウィンドウでなくなった	Event
p2ce-menuselect	メニューが選択された	CommandEvent (ウィンドウに送られ、メニュー項目には送られません)。
p2ce-command	メニューが選択されたか、プッシュ ボタンがクリックされた	CommandEvent (ウィンドウに送られ、メニュー項目には送られません)。

18.3.2 Gadgetイベント

以下の表は、ウィンドウに送信されるすべてのイベントを示したものです。イベントは数値ですが、この表では、¥netexpress¥base¥source¥guiclディレクトリにあるp2cevent.cpyの中で定義されているシンボリック識別子で示します。「イベント オブジェクト」欄は、イベントにより呼び出されるメソッドに渡されるイベントのタイプを示します。イベントのタイプに関する詳細な情報について、イベント オブジェクトを問い合わせることができます。

イベント	説明	イベント オブジェクト
p2ce-gainfocus	ガジェットがフォーカスを取得した	Event
p2ce-losefocus	ガジェットがフォーカスを失った	Event

"accessSystemEvents"メッセージを送信した場合、ガジェットはWindow イベント（Mouseイベントを含む）も受け取ります。

18.3.3 List Box、Selection Box,、TreeViewイベント

以下の表は、リストボックス(List Box)、選択ボックス(Selection Box)、およびツリービュー(Tree View)に送信されるすべてのイベントを示したものです。イベントは数値ですが、この表では、¥netexpress¥base¥source¥guiclディレクトリにあるp2cevent.cpyの中で定義されているシンボリック識別子で示します。「イベント オブジェクト」欄は、イベントにより呼び出されるメソッドに渡されるイベントのタイプを示します。イベントのタイプに関する詳細な情報について、イベント オブジェクトを問い合わせることができます。

これらのオブジェクトはGadgetからサブクラス化されるため、Gadgetイベントも受け取ります。

イベント	説明	イベント オブジェクト
p2ce-accept	項目の上でダブル クリックした	Event
p2ce-select	項目の上でシングル クリック（強調表示）した	SelectEvent

18.3.4 Scrollbarイベント

以下の表は、Scrollbarに送信されるすべてのイベントを示したものです。イベントは数値ですが、この表では、¥netexpress¥base¥source¥guiclディレクトリにあるp2cevent.cpyの中で定義されているシンボリック識別子で示します。「イベント オブジェクト」欄は、イベントにより呼び出されるメソッドに渡されるイベントのタイプを示します。イベントのタイプに関する詳細な情報について、イベント オブジェクトを問い合わせることができます。

これらのオブジェクトはガジェットからサブクラス設定されるため、ガジェット イベントも受け取ります。

イベント	説明	イベント オブジェクト
p2ce-lineup	1行上に移動した（エンドユーザがスクロール バーの上/下矢印をクリックした）。Thumb(つまみ)は移動していません。スクロールバーはこのイベントを受け取ると、アプリケーションが1行の距離として定義した間隔だけ移動します。	ScrollEvent
p2ce-linedown	1行下に移動した（エンドユーザがスクロール バーの上/下矢印をクリックした）。Thumb (つまみ)は移動していません。スクロールバーはこのイベントを受け取ると、アプリケーションが1行の距離として定義した間隔だけ移動します。	ScrollEvent
p2ce-pageup	1ページ上に移動した（エンドユーザがスクロール バーのつまみの上をクリックした）。つまみは移動していません。スクロールバーがこのイベントを受け取ると、アプリケーションが1ページの距離として定義した分だけ移動します。	ScrollEvent
p2ce-pagedown	1ページ下に移動した（エンドユーザがスクロール バーのつまみの下をクリックした）。つまみは移動していません。スクロールバーがこのイベントを受け取ると、アプリケーションが1ページの距離として定義した分だけ移動します。	ScrollEvent

p2ce-verttrack	スクロールバーのつまみが新しい位置にドラッグされた	ScrollEvent
p2ce-vertposition	特定の位置にスクロールした。縦のスクロールバーを移動した後で受け取ります。	ScrollEvent
p2ce-vertend	スクロールバーのつまみを端に移動した	ScrollEvent
p2ce-lineleft	1行左に移動した（エンドユーザがスクロール バーの左/右矢印をクリックした）。つまみは移動していません。スクロールバーがこのイベントを受け取ると、アプリケーションが1行の距離として定義した間隔だけ移動します。	ScrollEvent
p2ce-lineright	1行右に移動した（エンドユーザがスクロール バーの左/右矢印をクリックした）。つまみは移動していません。スクロールバーがこのイベントを受け取ると、アプリケーションが1行の距離として定義した分だけ移動します。	ScrollEvent
p2ce-pageleft	1ページ左に移動した（エンドユーザがスクロール バーの左/右矢印をクリックした）。つまみは移動していません。スクロールバーがこのイベントを受け取ると、アプリケーションが1ページの距離として定義した分だけ移動します。	ScrollEvent
p2ce-pageright	1ページ右に移動した（エンドユーザがスクロール バーの左/右矢印をクリックした）。つまみは移動していません。スクロールバーがこのイベントを受け取ると、アプリケーションが1ページの距離として定義した分だけ移動します。	ScrollEvent
p2ce-horiztrack	スクロールバーのつまみが新しいポジションにドラッグされた	ScrollEvent
p2ce-horizposition	特定の位置にスクロールした。 横方向のスクロールバーを移動した後で受け取ります。	ScrollEvent
p2ce-horizEnd	スクロールバーのつまみを端に移動した	ScrollEvent

18.3.5 UpDown イベント

以下の表は、ウィンドウに送信されるすべてのイベントを示したものです。イベントは数値ですが、この表では、¥netexpress¥base¥source¥guiclディレクトリにあるp2cevent.cpyの中で定義されているシンボリック識別子で示します。

「イベント オブジェクト」欄は、イベントにより呼び出されるメソッドに渡されるイベントのタイプを示します。イベントのタイプに関する詳細な情報について、イベント オブジェクトを問い合わせることができます。

イベント	説明	イベント タイプ
p2ce-lineup	上へ1行移動した	ScrollEvent
p2ce-linedown	下へ1行移動した	ScrollEvent

18.3.6 Button**イベント**

以下の表は、ボタンに送信されるすべてのイベントを示したものです。イベントは数値ですが、この表では、¥netexpress¥base¥source¥guiclディレクトリにあるp2cevent.cpyの中で定義されているシンボリック識別子で示します。これらのオブジェクトはGadgetからサブクラス化されるため、Gadgetイベントも受け取ります。「イベント オブジェクト」欄は、イベントにより呼び出されるメソッドに渡されるイベントのタイプを示します。イベントのタイプに関する詳細な情報について、イベント オブジェクトを問い合わせることができます。

イベント	説明	イベント タイプ
p2ce-clicked	ボタンがクリックされた	Event

18.3.7 Mouse **イベント**

以下の表は、マウス カーソルの下のオブジェクトに送信されるすべてのイベントを示したものです。これらのイベントは通常、Windowにのみ送信されますが、"acccessSystemEvents"メッセージをそれに送信する場合、Gadgetそれらにアクセスすることもできます。

イベントは数値ですが、この表では、¥netexpress¥base¥source¥guiclディレクトリにあるp2cevent.cpyの中で定義されているシンボリック識別子を示します。

イベント	説明	イベント オブジェクト
p2ce-mousemoved	マウスを移動した	MouseEvent
p2ce-button1down	ボタン1を押したままにした	MouseEvent
p2ce-button1up	ボタン1を離した	MouseEvent
p2ce-button2dblclk	ボタン2をダブルクリックした	MouseEvent
p2ce-button2down	ボタン2を押したままにした	MouseEvent
p2ce-button2up	ボタン2を離した	MouseEvent
p2ce-button2dblclk	ボタン2をダブルクリックした	MouseEvent

p2ce-button3down	ボタン3を押したままにした	MouseEvent
p2ce-button3up	ボタン3を離した	MouseEvent
p2ce-button3dblclk	ボタン3をダブルクリックした	MouseEvent

18.4 ビジュアル オブジェクトの作成

GUI クラス ライブラリは、デスクトップに表示できるそれぞれのタイプのグラフィカル オブジェクトに対して別のクラスがあります。 たとえば、Window クラス、Modal クラス、pushbutton クラスがあります。 Window クラスのインスタンスはウィンドウで、Modal クラスのインスタンスはモーダル(Modal)ダイアログ ボックスで、pushbutton ボタン クラスのインスタンスはプッシュボタンです。

提供された基本クラスをサブクラス化し、希望の動作を追加することにより、インタフェース（ウィンドウやダイアログボックスなど）にもっと複雑なオブジェクトを作成します。 簡単なオブジェクト（プッシュボタンや入力フィールドなど）であるほど、通常は、提供された基本クラスのインスタンスだけになります。

ダイアログボックスに希望のすべてのコントロールやガジェットを追加するには、次の2つの基本テクニックの一つを使用することができます。

- 静的
  提供されたDialog Editorを使用して、Windowsのリソースとして作成します。 これは、「About」ボックスを作成するために、テンプレート アプリケーションにより使用される方法です。
- 動的
  ランタイム時にガジェットを作成する Object COBOL コードを作成します。

1番目の方法のほうがより単純で高速です。しかし、2番目の方法はランタイム時に動的にインタフェースを変更するアプリケーションを作成できるという利点があります。

ウィンドウがリソースから作成できないとき、アプリケーションのメイン ウィンドウを動的にコーディングします。テンプレート アプリケーションの一部として提供されているWindowTemplateクラスを編集することは、自分独自のウィンドウを作成し始める場合の最も簡単な方法です。

詳細については、以下の項を参照してください。

- 18.4.1 ウィンドウとダイアログボックスの動的な作成
- 18.4.2 リソースからのダイアログボックスの作成

18.4.1 ウィンドウとダイアログボックスの動的な作成

ログボックスのそれぞれのタイプに対してサブクラスを作成することで、こ動的にウィンドウやダイアログボック

スを作成することは、ウィンドウがどのように表示されるか(サイズやタイトル、追加するガジェットなど)を正確に決めるコードを作成することを意味します。表示したいウィンドウやダイアれを行います。

ウィンドウやダイアログボックスを動的に作成するための最も簡単な方法は次のとおりです。

1. Windowか、ModalとModeless（ダイアログボックスの場合）からサブクラス化します。
2. 各サブクラスについて、オブジェクトの表示形式を決めるコードを追加、つまりウィンドウまたはダイアログボックスのためのタイトル、サイズ、位置、コントロール（プッシュボタン、メニュー、入力フィールドなど）を設定するインスタンス初期化コードを作成し、インスタンス メソッド"initialize"を上書きします。 スーパークラスにメッセージ"initialize"を送信して、"initailize"メソッドを起動しなければなりません。
3. ウィンドウやダイアログボックスを使用したいときは、作成したサブクラスに"new"メッセージを送信することにより(次の例を参照)、そのインスタンスを作成します。 そのオブジェクトの親を指定することが必要です。

新しいGUIオブジェクトの作成例

```
invoke BankWindow "new" using oo-desktop
                    returning aBankWindow
```

上のコードでOO-DESKTOPはウィンドウの親です。ウィンドウまたはダイアログボックスの親は、他のウィンドウまたはダイアログボックスか、オペレーティングシステムのデスクトップです。オペレーティングシステムはOO-DESKTOPと表され、EventManagerによって初期化できるようにOBJECT REFERENCE EXTERNALとして宣言する必要があります。

"new"メソッド（付属のGUIクラスからサブクラス化する場合に実現される）は、GUIオブジェクトの新しいインスタンスを作成し、そのインスタンスにメッセージ"initialize"を送ります。"initialize"メソッドにはインスタンスを初期化するためのすべてのコードを入れます。このメソッドは"new"メソッドによって起動されます。

"initialize"メソッドは次のインタフェースを持っていなければなりません。

```
method-id. "initialize".
linkage section.
  01  lnkParent       object reference.
procedure division using lnkParent.
invoke super “initialize” using lnkParent
```

***このインスタンスを初期化するコード**

```
    exit program.

  end method "initialize".
```

"initialize"メソッドの最初の行は、スーパークラスにメッセージ"initialize"を送ります。これによって、GUIオブジェクトのための基本的な初期化が行われます。これには、GUIオブジェクトをその親の子として登録する操作が含まれます。

Window サブクラスの例

この例では、Windowのサブクラス(ExampleWindow)のためのスケルトン コードを示します。このコードを適合させて、必要な表示を持つウィンドウを生成することができます。"new" メッセージを送信してExampleWindowをインスタンス化するとき、ExampleWindowにある"initialize"メソッドが、Windowクラスにあるコードによって起動されます。

```
    class-id. ExampleWindow data is protected
            inherits from Window.


    object section.
    class-control.
       BankWindow is class "bwindow"
       Window is class "window"
        ...
       .
    working-storage section.
    copy "p2cevent.cpy".


    ...
    Object.
```

```
 object-storage section.

 ...
```

- "initialize" メソッドは、、ウィンドウの"new"メソッド(パラメータとしてこのウィンドウの親を渡す)により起動されます。

```
method-id. "Initialize".


local-storage section.
```

- このメソッドのデータ宣言

```
01 lsX            pic x(4) comp-5.

01 lsY            pic x(4) comp-5.

01 lsCx           pic x(4) comp-5.

01 lsCy           pic x(4) comp-5.

...


linkage section.

   01 lsParent    usage object reference.

procedure division using lsParent.
```

- スーパークラス ウィンドウの初期化コードを実行します。サブクラスのいずれかの初期化コードを実行する前にこれを実行しなければなりません。

```
   invoke super "initialize" using lsParent
```

- ウィンドウのサイズと位置を設定します。

```
   move 2 to lsX
```

```
        move 2 to lsY
        move 40 to lsCx
        move 10 to lsCy
        invoke self "setRectangle" using lsX lsY lsCx lsCy
```

- W012. ウィンドウのタイトルを付けます。

```
        invoke self "setTitleZ" using "A Window" & x"00"
```

- メニュー、サブウィンドウ、コントロールなどを追加するコードを作成します。
- イベントへの関係を登録するコードを作成します。

```
        ...
        exit method.
    end method "Initialize".
    end object.
    end class ExampleWindow.
```

BeepWindow（beepwin3.cbl）クラスとBankWindow（bwindow.cbl）クラスには、GUIオブジェクトを初期化するためのコード例がいくつか入っています。BeepWindowのための初期化コードについては、第5章「クラスプログラムを作るチュートリアル」を参照してください。GUIプログラミングについては、第9章「GUIプログラミングのチュートリアル」を参照してください。

18.4.2 ダイアログ ボックスの静的な作成

Dialog Editorを使用すると、画面上にダイアログ ボックスを、すべてのガジェットとともに描くことができます。 これをコンパイルおよびリンクし、アプリケーションがロード可能なWindowsリソースにすることができます。 アプリケーションは、ダイアログ ボックスをロードすると、操作する必要のあるガジェットのためのオブジェクト ハンドルを取得します。

各ガジェットは、リソース ファイル内のリソース ID で表されます。 これらの ID は、Dialog Editorにより、コピーファイルを通して利用可能になるようになります。

Object COBOLにあるWindows リソースを使用する最も簡単な方法は、次のとおりです。

1. Object COBOLの中でリソースを表している、Modal またはModelessのサブクラスを作成します。
2. ガジェットのためのオブジェクト ハンドルをリソースから取得するコードを追加して、インスタンス メソッド "create"を上書きします。 これを行うには、"getObjectFromId"メッセージを送信します。 作成した"create" メソッドは、スーパークラスに"create"メッセージを送信して、起動しなければなりません。
3. ダイアログ ボックスを使用したいときは、作成したサブクラスに"fromResource"メッセージを送信することにより、そのインスタンスを作成します (この処理によって、リソース ファイルからリソースがロードされます)。

リソースをロードするためのグラフィカル サブクラスの例

この例では、Modal (モーダル(modal)ダイアログ ボックスを作成するためのクラス)のサブクラス(ExampleDialog)のためのスケルトン コードを示します。このコードを適合させて、必要な表示を持つダイアログ ボックスを生成することができます。

このサブクラスは、"create" メソッドを上書きしてオブジェクト ハンドルを取得し、コントロールとガジェットのためにマッピングするイベントをセットアップします。

```
class-id. ExampleDialog

      data is protected

      inherits from Modal.


 object section.

 class-control.

    BankDialogBox is class "bankdbox"

    ExampleDialog is class "xdial"

    ...

   .

working-storage section.
```

- XDIAL.CPY は、Dialog Editorで生成されるコピーファイルであり、このダイアログ ボックスにあるすべてのリソースの ID を定義しています。

```
copy "xdial.cpy".


 Object.
 object-storage section.
 01 anEntryField    object reference.
 ...


 method-id. "create".


 local-storage section.
 01 lsResourceId         pic x(4) comp-5.
 ...
 procedure division.
```

- スーパークラス作成コードを実行します。

```
    invoke super "create"
```

- ガジェットとコントロールのためのハンドルを取得します。EF-NAME は XDIAL.CPY で定義されているリソースIDの1つです。

```
    move EF-NAME to lsResourceId
    invoke self "getObjectFromId" using lsResourceId
                          returning anEntryField
    ...
```

- ガジェットとコントロールのためのイベントのマッピングをセットアップします。

- …

```
        exit method.

    end method "create".

    end object.

    end class namebox.
```

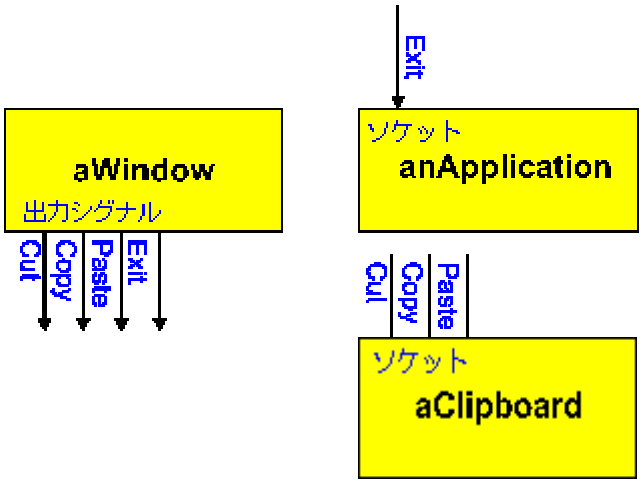
関連項目

- 25.3「GUIアプリケーションのテンプレート」

第19章 コンポーネントのフレームワーク

この章では、コンポーネントのフレームワークについて説明します。

19.1 概要



コンポーネントのフレームワークでは、オブジェクトを、プラグ接続可能なコンポーネントと同じように取り扱うことができる機構を提供します。

コンポーネントとしてオブジェクトを定義すると、次のことが可能になります。

- コンポーネントが送信できる出力シグナルを定義する。
- 他のコンポーネントからのシグナルを受信できる入力ソケットを定義する。 任意のコンポーネントに対して複数のソケットを定義することができます。
- 他のコンポーネントの入力ソケットに、コンポーネントの出力を接続する。 任意の数の入力ソケットにコンポーネントを接続することができます。

コンポーネントの入力ソケットでは、他のコンポーネントの出力に接続されたとき受信する、すべてのシグナルの動作を定義する必要はありません。

コンポーネントがシグナルを送信するときには、特定のオブジェクトに対してそれを送信するわけではありません。それは、シグナルを送信する回路ボードのチップと同じように、シグナルを送信した出力ピンに接続された任意の他のデバイスもそれを受信します。 シグナルを受信するために接続されたデバイスが無い場合、シグナルは何も行いませんが、回路ボードの故障の原因にはなりません。

メッセージでは、状況が異なります。メッセージでは、メッセージを送信するときに、一つの宛先オブジェクトを指定しなければなりません。 有効なオブジェクトを指定しなかった場合、またはメッセージを理解しないオブジェ

クトを指定した場合は、アプリケーションが中断します。 このコンポーネント機構を使用すると、共通ビルディング ブロックとして動作するオブジェクト（または共同オブジェクトのグループ）を定義することができます。 その後、コンポーネントのグループを互いにプラグ接続することにより、アプリケーションを構築します。

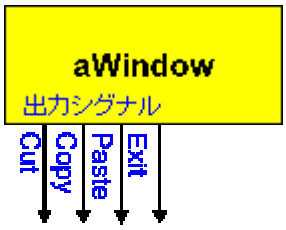
シグナルとソケットの使用方法

1. コンポーネントにオブジェクトを作成し、発生するシグナルを定義します。 出力コンポーネントが定義されました。
2. コンポーネントに他のオブジェクトを作成し、入力ソケットを作成します。 ソケットが認識する入力シグナルを定義し、続いて、シグナルの受信時に呼び出されるメソッドを定義します。 入力コンポーネントが定義されました。
3. 2 つのコンポーネントを互いに接続します。
4. 出力コンポーネントが入力ソケットにより定義されるシグナルを送信するたびに、対応するメソッドが入力コンポーネントで呼び出されます。

次の例を参照してください。以降の対応する項に説明があります。

- 出力シグナルの定義例
- 入力ソケットの定義例
- コンポーネント接続例
- シグナルの送信例
- シグナルのコピーファイル例

19.2 出力シグナルの定義



コンポーネントでは、生成するすべてのシグナルに対して名前を定義しなければなりません。 シグナルの名前には、最大32文字の任意の英文字で、ヌルで終了する文字列を指定できます。 特定のコンポーネントに対して定義されている出力シグナルは、それぞれ一意な名前を持たなければなりません。 オブジェクトに対して出力シグナルを定義する前には、それをコンポーネントにしなければなりません。

名前は、アプリケーション全体を通して一意であることが必要です。 多くの異なるコンポーネントに、同じ名前を持つシグナルを定義できます。 シグナルを受信する各コンポーネントは、接続したいコンポーネントのタイプが異なる場合は、それぞれ異なる入力ソケットを定義します。 したがって、異なるタイプのコンポーネントが同じ名前を持つシグナルを定義しても、名前の衝突はありません。

出力シグナルの定義例

この例では、ウィンドウ オブジェクト内のコードは、ユーザ イベントへの応答で一連のシグナルが送信されるように定義しています。シグナルの名前は、ヌルで終わるリテラル定数であり、この例ではコピーファイルでレベル 78 として定義されています。

- このオブジェクトが、シグナルを定義および送信できるようにします。

```
make component
```

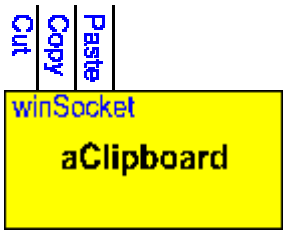
- シグナルを定義します。

```
define signal new-signal

define signal open-signal

define signal save-signal

define signal exit-signal

define signal about-signal

define signal help-signal
```

このコードは base.if で定義されたボキャブラリを使用しています。MAKE COMPONENTおよびDEFINE SIGNAL動詞を使用するには、プログラムの先頭にコピーファイルとしてこれを挿入する必要があります。

コピーファイルのコードの例については、19.5の「シグナルのコピーファイル例」を参照してください。

19.3 入力ソケットの定義



他のコンポーネントからのシグナルを受信するには、そのコンポーネントに対応する入力ソケットを定義しなけれ

ばなりません。 一意の名前を持つ入力ソケットをそれぞれ識別します。 名前には、最大 32 文字の長さの任意の英文字で、ヌルで終了する文字列を指定できます。 それぞれが違う名前である限り、コンポーネントに対して希望の数の入力ソケットを定義することができます。 異なるコンポーネントには、同じ名前の入力ソケットを指定することができます。

入力ソケットを定義したら、コンポーネントのメソッドに対して、その入力ソケットで受信することが期待されるすべてのシグナルをマップする必要があります。

入力ソケットの定義例

この例では、2 つの入力シグナルに応答する入力ソケットを定義します。ここでは、シグナルのコピーファイルで定義されているレベル 78 のリテラル定数の項目を使用して、ソケットとシグナルの名前を命名します。

"defineInSignal" メソッドは、そのソケットを通じて発生するシグナルのsignal-idを、コンポーネント内の特定のメソッドへリンクします。

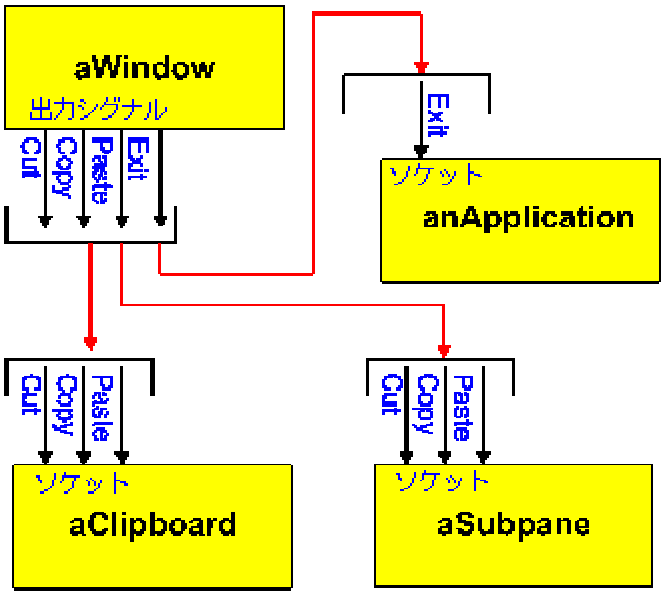
```
make component

declare socket mainApplicationSocket *> ソケットを宣言します。

map signal fileOpen-signal from mainApplicationSocket

                        to method z"onSignalFileOpen"

map signal fileOpenfailed-signal from mainApplicationSocket

                        to method z"onSignalOpenFailed"
```

このコードは base.ifで定義されたボキャブラリを使用しています。MAKE COMPONENTおよびDEFINE SIGNAL動詞を使用するには、プログラムの先頭にコピーファイルとしてこれを挿入する必要があります。

コピーファイルのコードの例については、19.5の「シグナルのコピーファイル例」を参照してください。

ボキャブラリの使用方法については20.2.3「ボキャブラリ定義された文や関数の使い方」の項を参照してください。

19.4 コンポーネントの接続



出力シグナルと入力シグナルは、互いに接続されているときにのみ動作します。 任意の数の入力ソケットに対して、1つの出力コンポーネントを接続することができます。また、任意の1つの入力ソケットを、複数のコンポーネントの出力に接続することができます。

コンポーネント接続例

この例では、入力コンポーネントで定義されているソケットを通じて、2つのコンポーネントが互いに接続されています。そのソケットの名前は、コピーファイルにあるレベル78データ項目によって命名されます。

```
connect wsWindow to wsApplication at mainWindowSocket
```

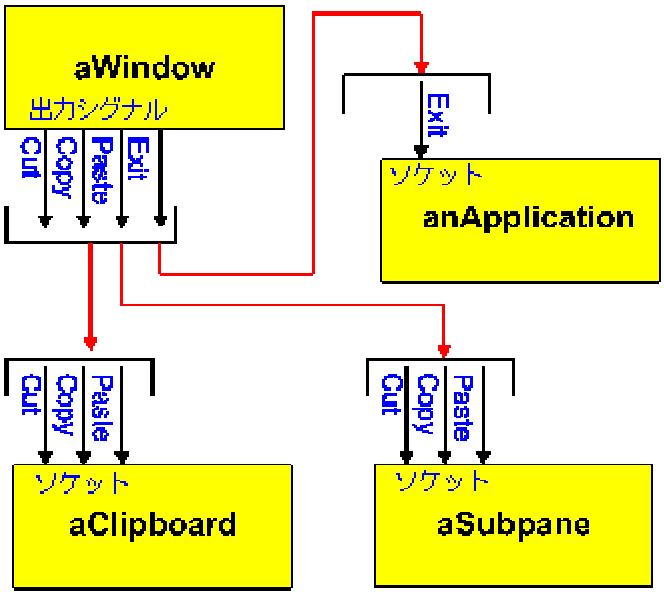
このコードは base.if で定義されたボキャブラリを使用しています。CONNECT動詞を使用するには、プログラムの先頭にコピーファイルとしてこれを挿入する必要があります。

コピーファイルのコードの例については、19.5の「シグナルのコピーファイル例」を参照してください。

ボキャブラリの使用方法については、20.2.3「ボキャブラリ定義された文や関数の使い方」の項を参照してください。

19.5「シグナルの送信」を参照してください。

19.5 シグナルの送信



コンポーネントはシグナルを送信するとき、そのシグナルを認識するような入力ソケットに接続されている場合、その入力ソケットで定義されているメソッドを呼び出します。 種々のコンポーネント上にある別々の入力ソケットに接続されているコンポーネントがある場合は、そのシグナルを認識するすべての入力ソケットが、メソッドを呼び出します。 メソッドはコンポーネントが互いに接続された順序で呼び出されます。

オプションとして、一つのオブジェクト ハンドルをシグナルと共にパラメータとして送信することができます。 これにより、シグナルと共に一部のデータの送信を可能にします。 データを要求するシグナルを送信したい場合がありますが、その場合は、受信した側のコンポーネントは、データを送信するためのシグナルを定義することが必要となります。

例えば、インタフェース コンポーネントは、"retrieve-accounts"シグナルを送信することにより、顧客の預金口座のリストを表示するように要求します。 検索コンポーネントは、このシグナルを受信し、データを見つけると、データとしての預金口座情報と共に"display-accounts"を送信します。 NetExpress GUI アプリケーション テンプレートは、SymbolTablesを使って、シグナルと共にデータを搬送します。

シグナルの送信例

この例では、コンポーネントがシグナルを送信します。シグナルの名前は、コピーファイルでレベル 78 として定義されています。パラメータ lsDataBlock には、シグナルと共にデータとして送信されるオブジェクト ハンドルが含まれています。

```
signal open-signal lsDataBlock
```

このコードは base.if で定義されたボキャブラリを使用しています。SIGNAL 動詞を使用するには、プログラムの先頭にコピーファイルとしてこれを挿入する必要があります。

コピーファイルのコードの例については、次の「シグナルのコピーファイル例」の項を参照してください。

シグナルのコピーファイル例

この例では、ウィンドウ用の一連の出力シグナルを定義するコピーファイルを示します。また、ソケットの名前も定義します。これは、これらのシグナルに対するインタフェースを定義したいコンポーネントで使用されます。

ヒント

必要なシグナルの名前のリテラル定数はすべて、コピーファイルでレベル 78 データ項目として定義し、常にコピーファイルにある名前によってシグナルを参照することをお勧めします。 プログラムをコンパイルするとき、シグナルのデータ名を間違って入力した場合に、コンパイル タイム エラーが発生します。コードの中でリテラル定数を直接使用している場合は、シグナルの名前を間違って入力してもランタイム時まではエラーが発生しません。

- ----------
- メイン アプリケーション ウィンドウ用のシグナルファイルのテンプレート
- ----------

```
78 exit-signal         value z"001".

78 about-signal        value z"002".

78 cut-signal          value z"003".

78 copy-signal         value z"004".

78 paste-signal        value z"005".

78 window-socket       value z"winSocket".
```

関連項目

- 入力ソケットの定義例

第20章 要件ベースのボキャブラリ

要件ベースのボキャブラリを使うと、作成したオブジェクトで使用するための新しいCOBOL文や関数を定義することができます。この章では、新しい文と関数を定義するための構文について説明します。

20.1 概要

INVOKE文に代わるメッセージ送信機構として、新しいCOBOLの文や関数を作成することができます。例えば、新しい文CREDITを作成すると、次のように記述することができます。

```
credit aBankAccount with amount
```

この構文をプログラムで使用するには、プログラムまたはクラスの最初にクラスインタフェース定義を入れる必要があります。インタフェース定義は、外部クラスによって宣言します。外部クラスには、クラスの構造があり、すべてのクラスとクラスのインスタンスメソッドを定義していますが、メソッドのためのコードは入っていません。

規定により、外部クラスは.ifファイルに保存され、ボキャブラリを使うプログラムはクラス定義を入れるためのCOPY文を使用します。COPY文はCLASS-ID見出しより前になければなりません。

20.2 要件ベースのボキャブラリの使い方

要件ベースのボキャブラリは、次の3つの手順により使用します。

1. 1つまたは複数のオブジェクトに関するボキャブラリを定義します。各オブジェクトのボキャブラリは、外部クラスで定義されます。
2. ボキャブラリによってオブジェクトを起動する各プログラム（またはクラス）の最初に外部クラスを入れます。
3. ボキャブラリによって定義される文や関数を使って、オブジェクトにメッセージを送ります。

多くの場合、手順1は、手順2と3とは異なるプログラマが担当します。一方のプログラマグループには、組織独自のクラス ライブラリのためのオブジェクトの設計タスクが与えられ、もう一方のプログラムグループは、クラス ライブラリのオブジェクトをビルディング ブロックとして使って業務アプリケーションを作成します。

クラス ライブラリの設計者は、クラス ライブラリ内のオブジェクト用のボキャブラリの定義（上記の手順1）を担当し、ボキャブラリの一貫性と可読性を保証します。アプリケーション構築者は、開発作業を単純化し、アプリケーションの可読性を向上させるためにボキャブラリを使用します（上記の手順2と3）。

20.2.1 ボキャブラリの定義

ボキャブラリはクラスを基準として定義します。各ボキャブラリは、外部クラスによって定義されます。外部クラ

スはメソッドインタフェースのセットを定義するプログラムですが、コードは含んでいません。

20.2.1.1 外部クラスの構造

外部クラスは、他のObject COBOLクラスと基本的に同じ構造です。

BankAccountの外部クラスの概略構造を次に示します。

```
class-id.BankAccount is external
          inherits from Base.
object section.
class-control.
  BankAccount is class "account"
  Base is class "Base"
  .

*class methods.
class-id."new"
*method interface definition.
end program "new".
...
*Object program
object.
*Instance methods
end object.
end class BankAccount.
```

BankAccountの外部クラスとBankAccountクラス自身は、次の点が異なるだけです。

- CLASS-ID段落ではクラスをEXTERNALとして定義しています。

- クラスまたはオブジェクトプログラムにはobject-storage節がありません。
- クラスの手続き部はありません。
- 各メソッドプログラムにはインタフェース定義だけが入っています。

20.2.1.2 メソッドインタフェース

メッソド プログラム インタフェースの定義では、新しいボキャブラリの部分のために次の項目を定義します。

- METHOD-ID段落
- LINKAGE SECTION
- PROCEDURE DIVISION見出し
- 文(動詞)シグネチャまたは関数シグネチャ
- END METHOD見出し

メソッド インタフェースの定義は一見普通のメソッドのようですが、記憶域は宣言されず、PROCEDURE DIVISION (手続き部) にコードはありません。

メソッドのためのインタフェースコード例

この例では、creditメソッドのためのインタフェースコードを示します。

```
method-id. credit is method.

linkage section.

01 lnkAmount               pic 9(10)v99.


procedure division using lnkAmount

 INVOKED AS ==Credit <self>[with]<lnkAmount> ==

     OR AS == deposit <lnkAmount>[in]<self> ==

.


end method credit.
```

上の例では、"credit" メソッドを起動するために 2 つの代替動詞を定義しています。次のように使用することができます。

```
credit anAccount with anAmount
```

または、次のように使用します。

```
deposit anAmount in anAccount
```

動詞または関数シグネチャは、メソッドの PROCEDURE DIVISION 見出しの一部として定義します。 定義は、INVOKED AS 句で始めます。 1 つのメソッドに、複数のシグネチャを定義することができます。 Vocabulary Generatorでは、任意の Object COBOL クラスのデフォルトのボキャブラリで、.if ファイルを作成するようになっています。詳細については、次の「要求ベースのボキャブラリを作成する方法」を参照してください。）

要求ベースのボキャブラリを作成する方法

1. Windows のタスクバーで [Start]、[Run] の順にクリックし、次のように入力します。

   ```
   runw vocabgen
   ```

2. [Vocabulary Generator] ダイアログ ボックスで、ボキャブラリを作成するクラスのパスとファイル名を入力し、[OK] をクリックします。

[Vocabulary Generator] は、クラスの全メソッドに対して、デフォルトの動詞と関数を持つ *filename*.ifを作成します。

3. 各 PROCEDURE DIVISION見出しの下にある二重等号（==）の間に定義されている構文を手直して、.if ファイルを編集し、必要な動詞または関数を定義します。 詳細については、上述の「メソッドのためのインタフェースコード例」を参照してください。

新しい構文定義の要素

新しい構文定義の要素には次のものがあります。

区切り子　　INVOKED AS段落は、動詞または関数のシグネチャ構文のはじめを示します。各シグネチャは二重等号"=="で囲みます。OR AS段落を使うと、1つのメソッドについて複数の定義を指定できます。

パラメータ　　角かっこ"< >"で囲みます。すべてのパラメータは、PROCEDURE DIVISION USING文の中で参照によって渡さなければなりません。

<SELF>は、送り手がメソッドのためのオブジェクト参照を入れる位置を示す特殊なパラメータです。<SELF>は、インタフェース定義の中に必ず1つ（しかも1つだけ）なければなりません。<THIS>と<OBJECT>を<SELF>の同義語として使用することもできます。

必須語　　そのまま指定しなければなりません。

ノイズワード　大かっこ"[ ]"で囲みます。ノイズワードは、新しい動詞や関数を読みやすくするために使用します。上の例では、次のどちらの方法でもcreditメソッドを使用することができます。

```
credit anAccount with aSumOfMoney

credit anaccount aSumOfMoney
```

デフォルトでは、メソッドインタフェース定義の動詞シグネチャで、以下のことが禁止されています。

- データ名の修飾として解釈できる位置における IN または OFの使用 (つまり、データ名が IN または OFの直前にあり、データ名が IN または OFの直後にある場合)。
- COBOL動詞の使用

この2つの禁止事項を両方とも禁止でないようにするには、コンパイラ指令OOCTRL(-Q)を設定します。デフォルトの設定はOOCTRL(+Q)です。

20.2.1.3 ユーザ定義の関数

ユーザボキャブラリによって新しい関数を作成することもできます。新しい関数は、プログラム内で使用した場合はCOBOLの固有関数と同様に取り扱うことができます。

例えば、次のようなコーディングが可能です。

```
compute interest = .06 * function balance (thisBankAccount)
```

キーワードfunctionは省略できるので、次のような記述も可能です。

```
compute interest = .06 * balance(thisBankAccount)
```

関数balanceの定義は次のようになっています。

```
method-id. "getBalance".

linkage section.

01 lsAmount          pic S9(10)v99.


procedure division returning lsAmount.

   INVOKED AS FUNCTION == balance (<self>) == .

end method "getbalance".
```

20.2.2 ボキャブラリの使用

ボキャブラリを使うには、それを定義する外部クラスのコピーが必要です。 規定により、これらは .if ファイルに保存されます。 プログラムの最初でCOPY文を使って、ソース コードに.if ファイルを入れてください。詳しい手順については、20.2.3「ボキャブラリ定義された文や関数の使い方」の項を参照してください。

NetExpressと共に提供されているMicro Focusのクラス ライブラリには、以下のタイプの操作に対するボキャブラリを持つ外部クラスが含まれています。

guibase.if イベント マッピング動詞

chararry.if 文字列操作動詞

symbtab.if シンボル テーブル(SymbolTable)操作動詞

sdiframe.if GUI アプリケーション フレームワーク動詞

20.2.3 ボキャブラリ定義された文や関数の使い方

クラスについてユーザ定義されたボキャブラリが存在する場合、どのようなObject COBOLクラスまたは手続きCOBOLプログラムでも簡単に使用することができます。

- **ボキャブラリ定義された動詞または関数を使用するには**

1. 新しい文を定義した外部クラスを自分のプログラム内にコピーします。複数のオブジェクトをそのボキャブラリ定義と共に使用する場合、それぞれの外部クラスをコピーしなければなりません。

   外部クラスは、プログラムの最初のPROGRAM-ID文やCLASS-ID文やIdentification Divisionより前に配置しな

ければなりません。

2. オブジェクトにメッセージを送るとき、INVOKE文とメソッド名のかわりに、オブジェクトについて定義されたボキャブラリを使用します。

3. プログラムをコンパイルするとき、mfooコンパイル指令を指定します。

ボキャブラリを使用する利点は、データ値がターゲットのメソッドによって期待されるフォーマットに変換されることです。例えば、次のようにコーディングできます。

```
credit anAccount with 5
```

プログラムの最初のインタフェース定義によって、コンパイラは数値5を、CREDIT 動詞で起動されるメソッドによって期待されるフォーマットに変換することができます。標準のObject COBOLを使って同じことを行おうとすると、次の2つの文が必要になります。

```
move 5 to anAmount

invoke anAccount "credit" using anAmount
```

要件ベースのボキャブラリの使用例については、第10章「要件ベースのボキャブラリのチュートリアル」を参照してください。

20.3 構文の定義

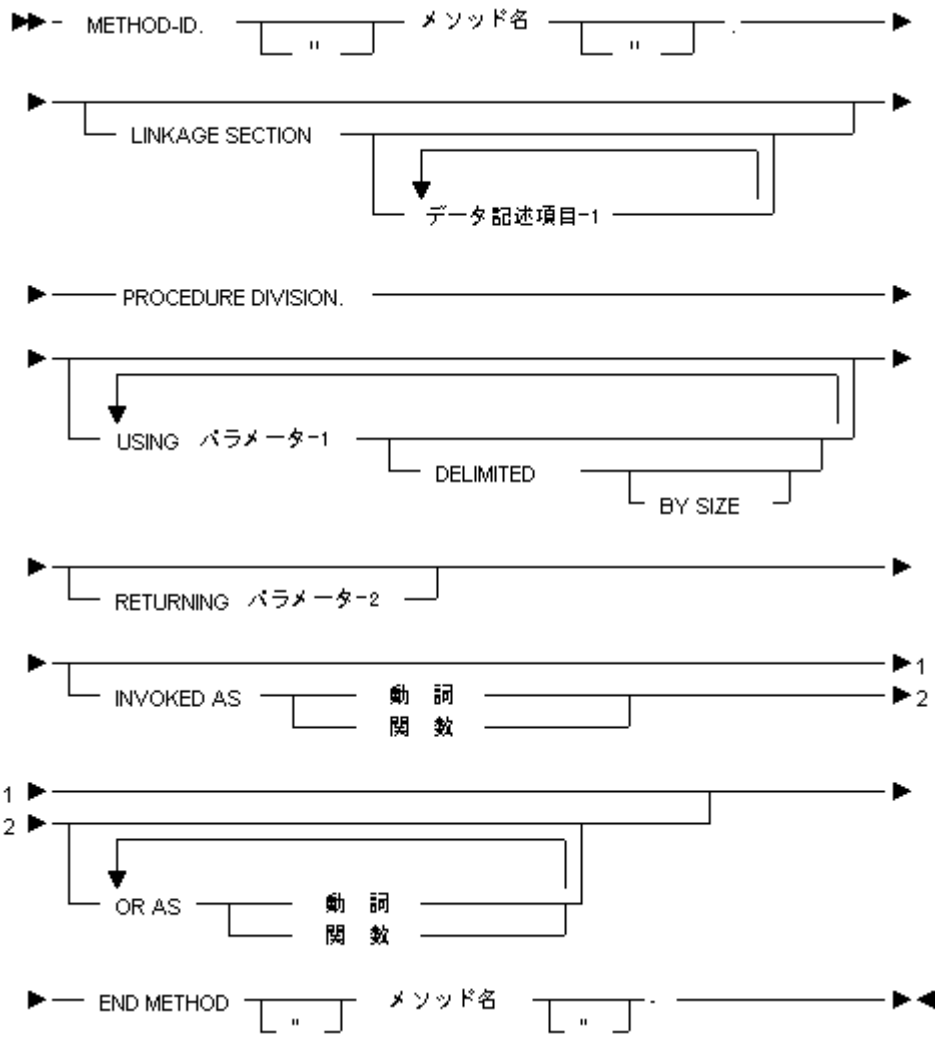
この節では、メソッドインタフェース定義を作成するための構文について説明します。

20.3.1 メソッドインタフェースの定義

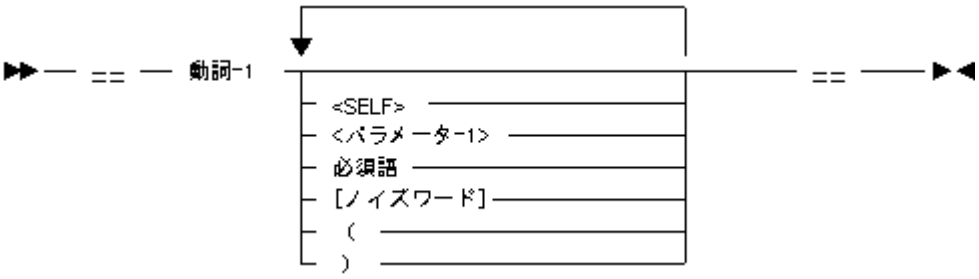
メソッドプログラムはコードを持っていません。メソッドインタフェース定義は、メソッドのためのパラメータ、パラメータを渡す方法、メソッド定義のための起動に使用できるクラスボキャブラリが定義された文または関数を定義します。

メソッドインタフェース定義は外部クラスに入れ子にされます。クラスメソッド定義は、クラス本体の中にあり、オブジェクトメソッド定義はオブジェクトプログラムの中にあります。

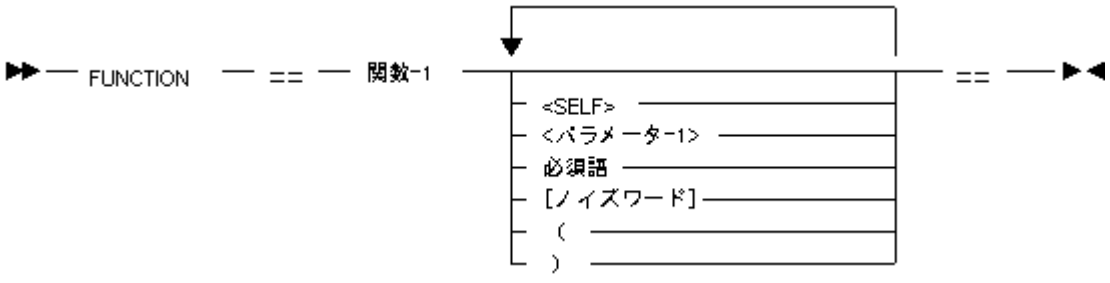
構文：



動詞のシグネチャは、



関数のシグネチャは、



規則：

1. DELIMITEDとDELIMITED BY SIZEは、英数字データ項目（PIC X(n)として定義されるもの）にだけ適用できます。両方とも文字列にx"00"を付加し、n+1バイトで渡します。

2. DELIMITEDは、メソッドを起動するコードによって送られるデータの終わりにx"00"を付加します。例えば、受け取るデータ項目がPIC X(10)と宣言され、メソッドを起動するコードが文字"ABCD"を送るとします。プログラムが受け取るデータは"ABCD0******"になります。ただし、0はx"00"（バイナリゼロ）、*は未定義文字です。

3. DELIMITED BY SIZEは、メソッドを起動するコードによって送られたバッファリングデータの終わりにx"00"を付加します。例えば、受け取るデータ項目がPIC X(10)と宣言され、メソッドを起動するコードが文字"ABCD"を送るとします。プログラムが受け取るデータは"ABCD******0"になります。ただし、0はx"00"（バイナリゼロ）、*は未定義文字です。

4. 動詞-1と関数-1は、有効なCOBOL識別子でなければなりません。COBOLの予約語は使用できません。

5. SELFは、ユーザ定義の構文内に必ず1つ（1つだけ）現れ、角かっこ"< >"で囲まなければなりません。SELFは、送り手が受け取るオブジェクトへのオブジェクト参照を入れる位置を示します。THISとOBJECTをSELFの同義語として使用することもできます。

6. パラメータ1（角かっこ"< >"で囲む）は01 linkage節のパラメータで、PROCEDURE DIVISION見出しのUSING段落になければなりません。

7. 関数のシグネチャを宣言するボキャブラリは、メソッドインタフェース定義の中でRETURNINGパラメータを指定しなければなりません。

8. 必須語は有効なCOBOL語でなければなりません。

9. ノイズワードは有効なCOBOL語で、大かっこ"[ ]"で囲まなければなりません。ノイズワードは読みやすくするための語で、文または関数を使ってメソッドを起動するために必須のものではありません。

10. かっこ"("と")"は、パラメータを大かっこ"[ ]"で囲んで、COBOLの固有関数のような関数を定義するのに使用します。

20.4 制限事項

1. 動詞-1と関数-1は、宣言されたデータ名と同じことがあります。この場合、COBOL Checkerは、常に文または関数の名前ではなくデータ名であると見なします。したがって、ユーザ定義のボキャブラリは、既存のプログラムのデータ項目に影響を与えずに拡張できます。しかし、新しい文または関数の名前を既存の手続き名と同じに定義すると問題が生じます。

2. ボキャブラリ定義は入れ子にできません。

3. ある定義が別の定義の拡張になることはできません。（定義Aが、定義Bとそれに続く1つまたは複数のパラメータから構成される場合、定義Aは定義Bの拡張になります。）

第21章 永続性

永続性は、実行中または実行後にアプリケーション内のすべてのオブジェクトの状態を保存するための機能です。この章では、アプリケーションの中での永続性の使い方について説明します。

21.1 概要

Object COBOLを使用するCOBOLプログラムは、アプリケーションの実行中または実行後にすべてのオブジェクトの状態を保存できます。PersistenceManagerクラスは、オブジェクトの状態の保存と復元に使用します。永続アプリケーションを再起動すると、オブジェクトが永続ファイルに記録された状態で復元されます。

永続性では、クラスプログラムのクラスとインスタンスのobject-storage節に宣言されたすべてのデータが保存されます。working-storage節またはlocal-storage節に宣言されたデータは、オブジェクトの属性とは見なされないために保存されません。

21.2 永続性の使い方

PersistenceManagerは、実行間でオブジェクトデータをCOBOL索引ファイルに保存します。COBOL索引ファイルはディスク上の2つの物理ライブラリからできています。

永続ファイルにはデフォルトでObjstore.datとObjstore.idxという名前が付いています。

永続ファイルはデフォルトで現在のディレクトリに書き込まれます。永続ファイルのデフォルトの名前と場所は、環境変数OODIRの設定によって変更できます。場所だけを変更するには、OODIRにパスを設定します。パス名の終わりは円記号"¥"にします。例えば、

```
set oodir=e:¥data¥
```

永続ファイルの名前はそのままで、ディレクトリe:¥dataに書き込みます。ファイル名を変更するには、OODIRにファイル名を設定します。

```
set oodir=mydata
```

永続ファイルはmydataとmydata.idxに書き込まれます。パスとファイルの両方を変更することもできます。例えば、

```
set oodir=e:¥data¥mydata
```

Object COBOL RTSも環境変数OODIRを使って実行開始時に永続ファイルの場所を探します。したがって、アプリケーションを再起動する場合、OODIRがアプリケーションを終了したときと同じ値になっているかを確認してください。

21.2.1 オブジェクトの保存

アプリケーションの実行中にいつでも、すべてのオブジェクトの状態を保存できます。永続アプリケーションは、オブジェクトを定期的に保存するように作成することも、アプリケーションの終了時にだけ保存するように作成することもできます。

永続性を使用すると、アプリケーション データ (従来のCOBOLファイルやデータベースで保存するデータ) だけでなく、インタフェースの状態も保存できます。このような保存は、アプリケーションの終了時にだけ行うことができます。これを行う場合には、アプリケーション内にある、ユーザ インタフェースの一部でないオブジェクトをすべて終了させてから、永続性を使用して残りのオブジェクトの状態を保存します。

アプリケーションの実行中または実行後にオブジェクトを保存するには、次の文を使用します。

```
invoke PersistenceManager "saveEverything"
```

すべてのオブジェクトが保存されます。

21.2.2 オブジェクトの復元

Object COBOL RTSは、Object COBOLアプリケーションの実行を開始するたびに、永続データが入ったファイルを検索します。Object COBOL RTSは、環境変数OODIRが示す場所を探します。

永続ファイルが見つかった場合、Object COBOL RTSはファイルに保存されたオブジェクトハンドルを復元します。オブジェクトがアプリケーションによって要求されると、データと共に復元されます。

永続ファイルから復元されたクラス用のクラス初期化コードは実行されません。永続アプリケーション内のクラス オブジェクトは、1回目の実行時にだけ初期化されます。永続アプリケーションを作成する場合、アプリケーションを開始するための短いトリガプログラム（Object COBOLクラスではない）を作る必要があります。

このプログラム自身は永続データを持っていません。このプログラムはクラスではないので、アプリケーションの実行を開始するためにクラスにメッセージを送ったり、あるいはアプリケーションの実行を開始するオブジェクトに対するオブジェクトハンドルを獲得する必要があります。プログラムが最初のメッセージを送ると、Object COBOL RTSは必要に応じて永続ファイルからオブジェクトのロードを始めます。

アプリケーションの1回目の実行では、永続データを使って起動する以後の実行時とは異なる方法でアプリケーションを初期化したい場合があります。 このプログラムは、PersistenceManagerクラスにメッセージ"isFirstSession"を送ることによってプログラムが1回目の実行かどうかを判断します。例えば、

```
invoke PersistenceManager "isFirstSession"
                        returning sessionStatus
```

sessionStatusはPic 9 Comp-x.として宣言されています。1回目の実行の場合（永続ファイルがない場合）、PersistenceManagerは1を返します。

21.2.3 永続バックアップファイル

PersistenceManagerには、永続ファイルをバックアップするための機能があります。次のコードは、最も新しい永続ファイルのバックアップを作成します。

```
invoke PersistenceManager "backupPersistenceFile"
```

現在の永続ファイルは、次のように名前が変更されます。

trigger.dat

trigger.idx

から

trigger.dt*n*

trigger.ix*n*

*trigger*は永続ファイルの名前、*n*は0～9の番号です。バックアップには0～9の番号が付けられ、バックアップ0が常に最新のバックアップです。

すべての永続ファイルを削除することもできます。これには、次のメッセージを送ります。

```
invoke PersistenceManager "eraseAllPersistenceBackupFiles"
```

21.2.4 永続性の例－Phonebook

Object COBOLには、Phonebookという永続アプリケーションの例が入っています。これは、名前と電話番号を格納するアプリケーションです。アプリケーションを中止すると、アプリケーション自身が永続ファイルに保存されます。アプリケーションを再起動すると、保存したすべての名前がインタフェースの状態と共に永続ファイルから復元されます。

- Phonebook**クラスをブラウズするには**

1. 現在のディレクトリをObject COBOLシステムをインストールしたドライブの¥cobol¥demo¥oopsに設定します（ここでは、デフォルトのセットアップでシステムをインストールしたものとします）。

2. 次のコマンド行を入力します。

```
browse phone.prj
```

Class Browserが起動し、Phonebookアプリケーション用のクラスがロードされます。PhonebookはGUIアプリケーションです。

Phonebookアプリケーションは次のクラスからできています。

PhoneApplication	アプリケーションを表し、Phonebookに格納されたレコードと情報の表示に使用するviewerクラスの間のリンクを提供します。
PhoneBook	PhoneBookオブジェクトを作成および初期化します。PhoneBookオブジェクトはレコードのコレクションを管理します。
Record	Phonebookの各項目についてレコードオブジェクトを作成します。
RecordViewer	レコードの詳細を表示するためのRecord details(view)ダイアログボックスを作成します。
RecordEditor	RecordViewerのサブクラスです。このクラスによって作成されたオブジェクトを使って、レコードの内容を変更し編集することができます。
PhoneWindow	PhoneBookの内容を表示するPhonebookウィンドウを作成します。

さらに手続き型COBOLプログラムが1つあります。

PhoneTrigger	これはアプリケーションを起動するためのものです。アプリケーションが1回目の実行かどうか、つまり永続ファイルがすでに存在しているかどうかをテストします。永続ファイルがない場合、アプリケーションを初期化するためのメッセージを送ります。永続ファイルがある場合、アプリケーションがすでに実行され、初期化されています。したがって、この場合はイベントを処理するインタフェースを起動するためにEventManagerにメッセージ"run"を送るだけです。

- **Phoneアプリケーションを試すには**

1. Class Browserの「COBOL」メニューから「すべてをチェック」を選択します。

   すべてのPhoneアプリケーションファイルがCOBOL Checkerにロードされます。

2. クラス／プログラムセレクタでPhoneTriggerを選択します。

3. 「COBOL」メニューから「実行」を選択します。

   Phone Bookウィンドウが表示されます。ここでは今回初めて実行するものと仮定します。永続ファイルはなく、ウィンドウには項目が表示されていません。

4. 「Add」ボタンを押します。

   レコードの追加ダイアログボックスが開きます。

5. 項目をタイプし、「OK」ボタンを押します。

   Phone Bookウィンドウに1番目のレコードが表示されます。デフォルトのPhone Bookウィンドウは名前リスト表示です。

6. 詳細リスト表示を見るために「File」メニューから「Detailed list」を選択します。

   2番目のPhone Bookウィンドウが開きます。これには、レコードにあるすべての詳細が表示されます。1番目のPhone Bookウィンドウが隠れてしまった場合、2番目のウィンドウを少し移動します。
   Phoneアプリケーションはデータの複数の表示をサポートしています。一度に任意の数の名前表示や詳細表示を見ることができます。
   レコードの表示、レコードの追加、レコードの編集の各ダイアログボックスも、一度にいくつでも開くことができます。
   レコードの編集ダイアログボックスを使ってレコード内のデータを変更し、「OK」ボタンを押すと、その

データに関する他のすべての表示（ウィンドウまたはダイアログボックス内）が自動的に更新されます。このような複数表示は、手続き型のアプリケーションよりもObject COBOLを使う方がはるかに簡単にプログラミングできます。各表示は、自分自身をデータに合わせて常に最新のものにするオブジェクトです。

7. Phonebookをしばらく試してみます。レコードをいくつか追加し、各種の表示を見てみます。

8. 操作が済んだら「File」メニューから「Suspend」を選択します。

   Phonebookは自分自身を永続ファイルに保存し、終了します。

ここでPhonebookを再起動できます。永続ファイルからは、データだけではなく、アプリケーションを中止したときの表示もすべて復元されます。

- **Phonebookを再起動するには**

1. クラス／プログラムセレクタでPhoneTriggerを選択します。

2. 「COBOL」メニューから「実行」を選択します。

   Phone Bookアプリケーションが中止直前の状態で復元します。

注: 同じディレクトリから別のObject COBOLアプリケーションを実行する前にObjstoreファイルの名前を付け直すか、削除するか、移動してください。この操作をしないと、Object COBOL RTSは永続ファイルを見つけ、新しく起動するアプリケーションにそのファイルを使おうとします。この場合、アプリケーションに予想できない結果が生じます。

21.3 永続性の制限事項

今回のバージョンの永続性には、次のような制限事項があります。

- 永続ファイルは、単一ユーザアプリケーションだけに適しています。

- 永続アプリケーションのクラスで宣言されたクラスまたはインスタンスのデータを変更した場合、再コンパイルしたアプリケーションを実行する前に永続ファイルを削除しなければなりません。変更後の永続データは、アプリケーションのクラスオブジェクトやインスタンスオブジェクトが使用するデータと一致しなくなり、アプリケーションはいずれかの時点で予想できない動作をします。

- 永続性機構は、不一致があっても警告しません。

- 永続性の実現方法は、将来のNetExpress リリースでは変更される可能性があります。将来の実現方法では、必ずしも既存のリリースとの下位互換性が維持されるとは限りません。

第22章 Object COBOLアプリケーションのデバッグ

この章では、Object COBOLアプリケーションをデバッグするときに役立つ情報を提供します。

22.1 概要

従来のCOBOLアプリケーションのデバッグに使うのとほとんど同じツールやテクニックが、Object COBOLアプリケーションにも使用できます。アプリケーション内のクラスは、従来のCOBOLプログラムと同じ方法でアニメートできます。

オブジェクト指向プログラムに固有の問題をデバッグするのに役立つ、特殊な機能もいくつかあります。

22.2 デバッグ機能

Object COBOLには、OOアプリケーションをデバッグするための次のような機能が用意されています。

従来の手続き型COBOLプログラミングでは、通常、データが割り当てられたメモリの場所がすべて分かるようになっていました。これは、データ項目が、プログラムで宣言されて表わされるためです。

OO プログラミングでは、新しいオブジェクトを作成する際、メモリが動的に割り当てられます。 Object Reference型のデータ項目(これはプログラムで宣言します)に、オブジェクトへのハンドルを格納します。オブジェクト ハンドルのレコードが、動的なオブジェクトの割り当てを使用している場合に手順から外れると、ランタイム時に予期しないエラーが発生します。以下の2つの問題の発生が考えられます。

- ハンドルが別のオブジェクトへのハンドルで上書きされたため、このオブジェクトへのハンドルが無くなっている。

  これは、メモリリークと呼ばれ、システム リソースを浪費します。

- 終了されているオブジェクトへのハンドルがある。

  Object COBOL RTS では、ハンドルを再使用してシステム リソースを節約します。このため、プログラム ロジック内の予期しないエラーの原因となり、そのハンドルが別のオブジェクトを指すのを止めてしまう可能性もあります。

このような問題を避けるには、以下の機能を使います。

- オブジェクトハンドルの再割り当てを防ぐための、OO RTSスイッチ(22.2.2 「オブジェクトハンドルの再割り当ての防止」参照)
- メモリリークを見つけるのに役立つ、オブジェクトカウンタ(22.2.4 「メモリリークの検索」参照)

- オブジェクト記憶保護ページ(22.2.5 「Object-Storageのガード ページ」参照)

Object COBOL RTS には、デバッキングの時間を節約する、メッセージ トレース機能も提供されています。 これにより、アプリケーションが送信したメッセージを初めから終わりまですべて見ることができます。アプリケーションがクラッシュまたはハングした場合でも、ログは完全なままです。

- アプリケーションによって送られたすべてのメッセージを示すログファイルを作成するための、メッセージトレース機能(22.2.3 「メッセージのトレース」参照)

デバッガを使って、クラス ライブラリ コードをアニメートすることもできます。

- クラス ライブラリのアニメート(22.2.1 「クラス ライブラリのアニメート」参照)

これらの機能について、次の各項で説明します。

22.2.1 クラス ライブラリのアニメート

NetExpressでは、GUI クラス ライブラリおよび基本クラス ライブラリのためのソース コードがすべて提供されています。 ソース コードは、デバッガを使ってステップ スルーすることができます。 ステップ スルーは、次のどちらかの理由から行う可能性があります。

- クラス ライブラリがどのように動作するかをもっとよく理解するため。
- 自分で作成したコードが、クラス ライブラリのメソッドを呼び出した後、失敗したため。

クラス ライブラリのアニメートは比較的簡単です。詳細については、次の「クラス ライブラリをアニメートする方法」を参照してください。

クラス ライブラリをアニメートする方法

1. 「プロジェクト」 メニューで、「プロパティ」 をクリックします。
2. 「プロジェクト プロパティ」 ダイアログ ボックスの一番下にある「クラス ライブラリをアニメート」で、アニメートするクラス ライブラリを次の中から選んでチェックします。

- Base class library (基本クラス ライブラリ)
- GUI class library (GUI クラス ライブラリ)
- OLE class library (OLE クラス ライブラリ)

3. 「OK」ボタンをクリックします。
4. アプリケーションのアニメートを開始します。

アプリケーションをステップ スルーしているとき、アニメートでチェックしたクラス ライブラリのメソッドを呼

び出す文の部分に差し掛かると、必ず、実行はクラス ライブラリ コードのアニメートに切り替わります。

22.2.2 オブジェクトハンドルの再割り当ての防止

OO RTSには、オブジェクトハンドルの再割り当てを防ぐためのオプションがあります。OO RTSのデフォルト動作では、最終化されたすべてのオブジェクトについてオブジェクトハンドルを再利用します。

オブジェクト参照が最終化された後にオブジェクトハンドルにメッセージを送るようなアプリケーションは、予想できない動作をすることがあります。この場合、古いハンドルが割り当てられている新しいオブジェクトにメッセージが送られてしまいます。

新しいオブジェクトが送られたメッセージを理解できない場合、"doesNotUnderstand"例外が発生し、エラーが起こったことがわかります。新しいオブジェクトがメッセージを理解した場合、メソッドを実行し、アプリケーションはもっと後で失敗するか、予想できない結果が生じます。

オブジェクトハンドルの再割り当てを防ぐには、変数OOSWを+dに設定します。

```
set oosw=+d
```

すでに存在しないオブジェクトに関するオブジェクトハンドルにメッセージを送った場合、OO RTSは次のエラーメッセージを表示します。

```
RTS 204 オブジェクト参照が不正です
```

注：
OOSWの+dは開発作業専用に用意された設定です。生産環境では+dを設定しないでください。OO RTSが割り当てるオブジェクトハンドルがなくなってしまうことがあります。こうなるまでにOO RTSが割り当てられるオブジェクトハンドルの数は、利用できるメモリによって異なります。

関連項目

- 11.5「オブジェクトの廃棄」

22.2.3 メッセージのトレース

アプリケーションが失敗したり、例外が発生した場所がうまく見つからない場合、メッセージトレースをオンにするとよいでしょう。これは、実行点がクラス ライブラリにあるときにエラーが起こる場合に特に便利です。 メッセージ トレースは、アプリケーションが送信したメッセージをすべて示すログ ファイルを生成します。トレース ファイルのフォーマットについては、次の「トレース ファイルのフォーマット」の項を参照してください。

備考：メッセージトレースをオンにすると、アプリケーションの実行が遅くなります。これは、送られたすべてのメッセージをファイルに書き込むためです。

メッセージ トレースをオンにする方法

IDE でメッセージ トレースを表示するには、以下のようにします:

1. 「オプション」メニューの「ログをトレース」をクリックします。
2. 「保存するメッセージ数」フィールドで、1〜1000 の間にある数字を入力します。

トレース ログは最新の *n* 個のメッセージを格納します。ここで、*n* はこのフィールドに入力した数字です。

3. 「表示」メニューの「デバッグウィンドウ−トレース ログ」をクリックして、プログラムをアニメートすると、トレース ログが表示されます。

メッセージトレースをオンにするには、アプリケーションを実行する前にOOSW環境変数にObject COBOL RTSの +t スイッチを次のように設定します。

```
set oosw=+t
```

トレース ファイルのフォーマット

メッセージ トレースがどのようにして開始されたかによって、OO メッセージ トレースからの出力方法は異なります。 1つは、 trace.log という名前のASCIIファイル(デフォルトではカレント ディレクトリに作成されます)への出力です。 もう1つは、ウィンドウへの出力で、これは「表示」 メニューの「トレース ログ」をクリックすると表示されます。 トレースは、アプリケーションが送信した1メッセージにつき1行表示されます。 次のリストでは、トレースで表示される各欄について説明しています。

トレースの作成と使用の詳細については、上述を参照してください。

Type of Resolve	INVOKEのターゲットは次の３つのコードに分類されます。
resolve	あるオブジェクトから他のオブジェクトへのINVOKE
resolveself	オブジェクトが自分自身を起動（INVOKE SELF...）
resolvetosuper	オブジェクトがそのスーパークラスを起動（INVOKE SUPER...）
Object reference	メッセージの受け手のオブジェクトハンドル
Message	INVOKEによって送られたメッセージ

Object type	次のいずれかのコードです。
I	インスタンスオブジェクト
C	クラスオブジェクト
M	メタクラスオブジェクト
Class of object invoked	起動された型のオブジェクトのクラス名
Class of implementor	メソッドの実現手段のクラス名
Stack level	メッセージのスタックレベルを示します。スタックレベルはメッセージが送られる度に大きくなり、メッセージが完了すると小さくなります。 例えば、メソッドAが他のメソッドを起動すると、スタックレベルは大きくなります。二番目のメソッドが実行を完了すると、スタックレベルは小さくなります。

22.2.4 メモリリークの検索

メモリリークは、アプリケーションが使い終わっていて、最終化されていないオブジェクトに割り当てられたメモリです。メモリリークを分析するには、アプリケーション内のオブジェクトの数を調べる方法があります。例えば、レコードを1つ追加すると24個のオブジェクトが増えるのに、レコードを1つ削除しても20個のオブジェクトしか取り除かれない場合、メモリリークの可能性があります。

Behavior に"getNumberOfObjects"メソッドを送ることで、アプリケーションで任意の時間に存在するオブジェクトの数を調べることができます。

オブジェクトの数を調べる方法

Behaviorクラスにメッセージ"getNumberOfObjects"を送ります。

例えば、

```
invoke Benhavior “getNumberOfObjcets” returning totalNumber
```

totalNumberはPIC X(4) COMP-5として宣言します。

22.2.5 Object-Storageのガード ページ

ガード ページスキャンは、あるタイプのメモリの破損の問題を検出するのに役立ちます。以下の2つのタイプのメモリのエラーを検出できます。

- Object-Storageデータ
- Baseクラスの "malloc"メソッドで割り当てられたメモリ

ガード ページを使って、以下のいずれかのトラップを行うことができます。

- メモリの先頭より前の部分の読み取りまたは書き込みをしようとする
- メモリの先頭より後の部分の読み取りまたは書き込みをしようとする

この種の問題は、部分参照を使ってデータをアクセスするとき、あるいはパラメータとしてメソッドにObject-Storageデータを渡すときに発生します。Object-Storageデータは、誤ったサイズを宣言されたLinkage Sectionのデータ項目を使って、メソッドにアクセスしようとします。

割り当てられているメモリおよび解放されているメモリへ、Baseクラスのメソッド "malloc" および "dealloc" を使用する場合、以下のタイプのエラーのトラップも行うことができます。

- 解放されたメモリを使おうとする
- メモリを2回解放しようとする

ガード ページをアクティブにした状態で実行しているときに、このタイプのエラーのいずれかが発生した場合は、ランタイム エラー 114 が発生します。プログラムをアニメートしている場合は、問題の原因となった文が強調表示された状態で、実行が停止します。

ガード ページをオンにする方法

ガード ページは、Object-Storageおよびメモリの割り当ての前または後のいずれかを設定することができます。

Object COBOL RTSの +g1 スイッチを使って、Object-Storageおよびメモリの割り当ての前にガード ページを設定します。

```
set oosw=+g1
```

Object COBOL RTSの +g2 スイッチを使って、Object-Storageおよびメモリの割り当ての後にガード ページを設定します。

```
set oosw=+g2
```

注:ガード ページを使って実行すると、アプリケーションによって使用されるメモリの量が増えます。 デバッグのときだけ使用してください。

22.3 トラブルシューティングのヒント

エラー メッセージの中には、OO プログラムで生成されるだけものと、OO と共通のものがあります。 トラブルシューティングのヒントを使うと、問題をより速く突き止めることができます。

この節では、以下の共通する問題について取り上げます。

- プログラムが見つからない
- invokeでのメモリ例外および保護侵害(22.3.2 「invokeでのメモリ例外および保護侵害」参照)
- 記号の再定義
- 予期しないアプリケーションの動作(22.3.4 「予期しないアプリケーションの動作」参照)

22.3.1 プログラムが見つからない

アプリケーションが、有効でないクラスを使用しようとすると、RTSはエラーメッセージ173(呼出すプログラムが見つからない)を出します。使用できるようにするには、現行ディレクトリまたは以下の環境変数が示すディレクトリ中のクラス用に、実行可能形式のファイルが必要です。

- COBDIR (.gnt形式用)

Windows

- PATH (Windows上の.dll形式)

OS/2

- LIBPATH (OS/2上の.dll形式)

22.3.2 invokeでのメモリ例外および保護侵害

メソッドから、またはメソッドにパラメータが渡される時、invokeで使用するデータ項目がメソッドによって期待されるものと一致しているかを確かめる必要があります。特に、メソッドがデータをRETURNINGパラメータへ転記しようとする場合、また、メソッドを呼出すinvoke文がRETURNINGパラメータを提供していない場合は、アプリケーションは保護侵害またはメモリ例外を起こします。

例えば、下記の"currentTime"メソッドは６バイト集団項目に時間を返します。

```
method-id. "currentTime".

linkage section.

01 lnkTime.

  03 lnkHours   pic xx.
```

```
  03 lnkMins    pic xx.
  03 lnkSecs    pic xx.
procedure division returning lnkTime.
move timeNow to lnkTime
exit method.
end method "currentTime".
```

次の文は、実行時に保護侵害またはメモリ例外を起こします。

```
invoke mainClock "currentTime"
```

関連項目

- 12.6.2 「メソッド データとパラメータ」

22.3.3 記号の再定義

OOプログラムは以下のような原因により、RTSエラー119(記号の再定義)を起こします。

- 同じオブジェクトに、同じメソッド名を２回定義している
- あるクラス名を、１つ以上のファイル名で定義している

一番目は、２つの同一のクラスまたは２つの同一のインスタンスメソッドを持っている時に起こります。クラス名とインスタンスメソッド名を同じにすることはできます。しかし例えば、クラスとインスタンスオブジェクトの両方に"initialize"メソッドを定義しているクラスプログラムを持っているかもしれません。

二番目は、異なるファイル名に対して、異なるプログラムのclass-control段落に定義されたクラスを持っている場合に起こります。ファイル名は大小文字を区別するので、このためにエラーとなっている可能性もあります。

例えば、プログラムAのclass-control段落が以下のようになっています。

```
class-control.
DateClass is class "Date"
...
```

そしてDateClassクラスのclass-control段落が以下のようになっています。

```
class-control.

DateClass is class "date"

...
```

実行時に、プログラムAがDateClassを起動しようとすると、OO RTSはエラー119を出します。提供するClass libraryとサンプルプログラムでは、すべてのファイル名は小文字で入力してあります。

22.3.4 予期しないアプリケーションの動作

Object COBOLの永続性機能（「第21章 永続性」を参照）は、他のアプリケーションが作成した永続ファイルをロードする時に、予期しない結果を招くことがあります。永続ファイルはobjstore.datまたはobjstore.idxという名前か、OODIR環境変数の設定によって名前が付けられます。

誤った永続ファイルで実行したアプリケーションの実際の動作は予想できません。 一般に、OO アプリケーションを実行しようとして、別のアプリケーションの永続ファイルがピックアップされた場合、永続ファイルを作成したアプリケーションが表示され実行されます。 ただし、予期しない時点で失敗することがあります。

例えば「第21章 永続性」で説明されたPhonebook用の永続ファイルを作成すると、ファイル名はobjstoreです。

永続ファイルが現行またはOODIRが指すディレクトリにある場合、他のObject COBOLアプリケーションを実行すると、OO RTSは現在あるPhonebook永続ファイルを開き、これを使おうとします。実行しているアプリケーションの代わりに、メインPhonebookウィンドウが現れ、予期しない時点で失敗します。

第23章 OLEコンテナおよびサーバのサポート

この章では、OLEコンテナおよびサーバのサポートについて説明します。

23.1 概要

OLE コンテナとしてアプリケーションを定義する場合、他の Windows アプリケーションで作成されたドキュメント オブジェクトを含むドキュメントを作成するために、それを使用することができます。 埋め込みオブジェクトをダブルクリックするとそれを作成するアプリケーションを起動し、その場で編集することができるようになります。 たとえば、COBOL アプリケーションに Excel ワークシートを埋め込むことができます。

NetExpress に含まれるOLE コンテナのテンプレートは、基本的なコンテナ アプリケーションを作成できるようにします。 また、コンテナ アプリケーションがどのようになるかを調べるためだけであれば、コンテナ アプリケーションを作成し、実行することもできます。 コンテナ テンプレートを使用する方法を調べるためには、第25章「OLEコンテナのテンプレート」を参照してください。

Object COBOL を使って OLE サーバも作成することができます。 OLE サーバはドキュメント オブジェクトまたは OCX コントロールになります。 OLE コンテナに OLE サーバを埋め込むことができます。 NetExpressでは、netexpress¥base¥demo¥oledemos¥controls にいくつかのデモ用の OCX コントロールが提供しています。OCX コントロールは、OLEサーバ技術の一つのアプリケーションです。

NetExpress の OLE サポートは、OLE クラス ライブラリにより指定されます。

注:NetExpress により指定された OLE コンテナおよびサーバのサポートを使用するためには、Object COBOL プログラミングの専門家で、Microsoft OLE 2 に関する知識を持っている必要があります。Microsoft Press の『OLE 2 Programmer's Reference』および『Inside OLE 』(Kraig Brockschmidt 著) は両方とも良い情報源です。

23.2 基本コンテナ サポートの追加方法

基本コンテナ サポートを追加するには、次の手順を行います。

1. OLE サポートを初期化します(23.2.1 「OLEサポートの初期化」参照)。
2. OLE ビュー クラスをコーディングします(23.2.2 「OLEビュークラスのコーディング」参照)。
3. OLE ドキュメント作成のメソッドをコーディングします(23.2.3 「OLEドキュメント作成のメソッドのコーディング」参照)。

23.2.1 OLEサポートの初期化

OLE サポートを初期化するには、次の手順を行います。

1. GUI クラス ライブラリをロードします。

   ```
   call "apigui"
   ```

   GUI アプリケーションにはすでにこの文が含まれています。

2. イベント マネージャのインスタンスを作成します。

   ```
   invoke EventManager "new"
   returning anEventManager
   ```

   パラメータ

   anEventManager OBJECT REFERENCE.

   GUI アプリケーションにはすでにこの文が含まれています。

3. "initializeOLE"メッセージをEventManagerインスタンスに送信します。

   ```
   invoke anEventManager "initializeOLE"
   returning anEventManager
   ```

   これにより OLE クラス ライブラリがロードされ、初期化されます。 ただ olecl を呼び出すだけでは、OLE クラス ライブラリを初期化することはできません。

23.2.2 OLEビュークラスのコーディング

OLE ビュー クラスを作成するには、次の手順を行います。

1. Subpaneのサブクラスを作成します。

   a. ブラウザ ウィンドウで右クリックし、「クラスの追加」 をクリックします。 これにより、「クラスの作成」ダイアログ ボックスが開きます。

   b. 「クラス名」フィールドで、クラスに付ける名前を入力します。（例： MyOleView）

   c. Subpaneを 「基底クラス」フィールドに入力します。

   d. 「ファイル名」フィールドでデフォルトを受け入れるか、または独自のファイル名を入力します。

   e. 「OK」ボタンをクリックして、新しいクラスを作成します。

サブクラスのインスタンスが、OLE ビュー オブジェクトです。 AbstractWindowのサブクラスである任意の GUI オブジェクトを ＯＬＥ ビューにすることができますが、ビューに独自のサブクラスを作成するほうが、柔軟性が高くなります。

2. Class-Control 段落に以下の項目を追加します。

   OLEClientItem にクラス "oleclnt"

   OLEDocument にクラス "oledoc"

   OLEDataObject にクラス "oledobj"

23.2.3 OLEドキュメント作成のメソッドのコーディング

OLE ドキュメント作成のメソッドをコーディングするには、次の手順を行います。

1. ビュー クラスの中に、インスタンス メソッド"newDocument" を作成します。

   a. OLE ビュー クラスのテキスト ウィンドウのツールバーにある「新規メソッド」ボタンをクリックします。

   b. 「メソッド名」フィールドにnewDocumentを入力します。

   c. 「OK」ボタンをクリックします。

2. "newDocument"は、最小限で以下の操作を実行する必要があります。

   a. OLE ドキュメントを作成し、アプリケーション フレームに置きます。 フレームは通常、ビューの親オブジェクトです。また、任意のGUI オブジェクトは、"getAncestor"メッセージを自身に送信して親を見つけることができます。

   b. このオブジェクト自身に"isOLEView" メッセージを送信して、このオブジェクトが、自身を OLE ビューに変換していないかどうかを調べます。

      - 変換していない場合は、ドキュメントをパラメータとして入れて、オブジェクト自身に"makeOLEView"メッセージを送信する必要があります。

      - 変換している場合は、オブジェクト自身に"setOLEdocument"メソッドを送信して、新しいドキュメントをオブジェクト自身に対して設定する必要があります。 そして、それ自身をアクティブ ビューにします。

3. オプションで、新しいドキュメントを作成する前に既存のドキュメントを確認するようなコードをメソッドに追加して、ドキュメントが作成されていたかどうかを示す成功コードを戻すことができます。

23.2.4 メソッドの例

これは、OLE ドキュメントを作成するための OLE ビュー サブクラスのメソッド例です。黒で表示されたコードは最低限必要なコードです。青で表示されたコードは、既存のドキュメントを確認し、それを保存する必要があるかを参照するための追加コードです。新規ドキュメントが問題なく作成された場合は追加コードは 1 を返し、それ以外は 0 を返します。

```
method-id. "new"

local-storage section.

01 lsBool                pic x comp-5.

01 lsBool                pic x comp-5.

01 lsDocFrame             object reference.


Linkage section.

01 lnkBool               pic x comp-5.


procedure division returning lnkBool. *> ドキュメントが作成されている

                                      *> 場合は、1を返す

   move 0 to lnkBool

   if osOLEDocument <> null          *> 既存のドキュメントを

                                      *> 調べます。osOLEDocument

                                      *> は、インスタンスObject-Storage

                                      *>  で宣言されています。

      invoke self "canClose" returning lsBool

                                      *> 現在のドキュメントが変更

                                      *> されたかどうかを調べます。

                                      *> これを調べるには、
```

```
                                *> "canClose"メソッドを提供する
                                *> 必要があります。
   if lsBool = 1
      invoke osOLEDocument "finalize"
                          returning osOLEDocument
   end-if
end-if
if osOLEDocument = null
   move 1 to lnkBool
   invoke self "getAncestor" returning lsDocFrame
                                *> このビューのための
                                *> フレームを取得します。
   invoke OLEDocument "new" using lsDocFrame
                       returning osOLEDocument
                                *> ドキュメントを作成します。
   invoke self "isOLEView" Returning lsBool
                                *> このオブジェクトが OLEビュー
                                *> であるかどうかを調べます。
   if (lsBool = 0)
      invoke self "makeOLEView" Using docFrame
                                   osOLEDocument
                                *> このオブジェクトをosOLEDocument上の
                                *> OLEビューにします。
      invoke self "enableDragDrop"*> オプション - OLE
```

```
                                    *> のドラッグ アンド ドロップ
      else
         invoke self "setOLEDocument" using osOLEDocument
                                    *> このビューのためのOLEドキュメント
                                    *> を設定します。
         invoke osOLEDocument "setActiveView" using self
      end-if
   end-if
   exit method.
end method "new".
```

第24章 OLEオートメーション

24.1 概要

オブジェクト リンクおよび埋め込み（OLE）は、Microsoft Windows の機能です。これにより、あるアプリケーションが、ActiveX オブジェクトとしてオペレーティング システムで登録された任意のアプリケーションへのメッセージをロードし送信することができます。 Micro Focus OLE オートメーション サポートを使うと、Object COBOLのプログラムおよびクラス（OLE クライアントとしてのObject COBOL）から、ActiveX オブジェクトへメッセージを送信することができます。 また、そのインスタンスがOLE サーバであり、それからActiveXオブジェクトを作成する、Object COBOLクラスを作成することもできます。

OLE オブジェクトは、Microsoft の Component Object Model (COM)を使って構築されています。 COM と、Object COBOL などの OO 言語用のモデルの間には、主に次のような2つの違いがあります。

- OLE は、継承をサポートしません。 新しいOLEクラスを作成するとき、既存のOLEクラスおよび非OLEクラスから機能を継承することはできません。
- OLE は、クラス メソッドおよびクラス データをサポートしません。

また、Microsoft は、分散バージョンのCOMもサポートしています。これはネットワーク上で機能し、DCOM と呼ばれています。DCOM を使うと、別のマシン上に常駐しているオブジェクトに対してメッセージを送信することができます。 OLE オートメーションの詳細については、オンライン マニュアルの『分散コンピューティング』(Oppubb03.htm)を参照してください。

注:Object COBOLを使って、OCXコントロールを作成することも可能ですが、この機能は現在マニュアルに記載されておらず、サポートもされていません。

24.2 OLEタイプ ライブラリの作成方法

OLE タイプ ライブラリを作成するには、以下の手順を行います。

1. タイプ ライブラリの必要なObject COBOLのOLE オートメーション サーバ クラスを、.int または .gnt形式にコンパイルします。
2. NetExpressの「ツール」メニューにある 「OLE Registry File Generator」をクリックします。

これにより、Object COBOLのRegistry File Generatorが開始し、OCREGというラベルのついたダイアログ ボックスとして表示されます。

3. 「追加」ボタンをクリックし、「追加するファイルを選択」ダイアログ ボックスを使って、タイプ ライブラリを作成する対象の、.intまたは.gnt形式版のObject COBOLのOLE オートメーション サーバーをすべて選択し

ます。

4. 「型ライブラリを生成」チェックボックスをクリックします。

5. 「OK」ボタンをクリックします。

注: この方法で作成したタイプ ライブラリは、コンパイル済みのバイナリ形式の.tlbファイルとして生成されます。タイプ ライブラリをソースの形式の.idlファイルとして作成するには、OLE オートメーション サーバーを作成する際に、「クラス ウィザード」を使ってタイプ ライブラリを生成します。

24.3 OLEオートメーション クラスの作成方法

OLE オートメーション クラスのインスタンスは ActiveX のオブジェクトであり、他の言語で作成されたクライアントを使って、また必要であればマシンまたはプロセスの境界でも呼び出すことができます。

詳細については、オンライン マニュアル『分散コンピューティング』の「OLE オートメーション」章を参照してください。

1. 「ファイル」メニューの「新規作成」をクリックし、「新規作成」ダイログ ボックスから「クラス」を選択して、クラス ウィザードを起動します。

2. 最初のダイアログ ボックスで「OLE オートメーション」ラジオ ボタンを選択し、「次へ」ボタンをクリックします。

ウィザードのダイアログ ボックスのプロンプトに従って、サーバーを作成します。

注:デュアル インタフェースを持つサーバーを提供するには、チェックボックスをクリックしてタイプ ライブラリを生成します。 タイプ ライブラリは、OLE オートメーション サーバー内のメソッドやプロパティを追加または削除したときに、必ず更新する必要があります。

24.4 オートメーション サーバのレジストリへの入力方法

クラス ウィザードを使ってOLE オートメーション サーバーを生成するときは、レジストリ入力ファイルが自動的に生成されるので、以下の手順 1 および 3 は省略できます。

1. NetExpressの「ツール」メニューにある 「OLE Registry File Generator」をクリックします。

これにより、Object COBOLのRegistry File Generatorが開始し、OCREGというラベルのついたダイアログ ボックスとして表示されます。

2. 次の各フィールドに入力します。

パラメータ

クラス	OLE クライアントがサーバーを参照するのに使用する名前です。サーバー名は、サーバー クラス(これがObject COBOLで作成されている場合はクライアント)のCLASS-CONTROL項目の右側に表示されていなければなりません。
コメント	サーバーの説明です。
コマンド行	OLE サーバーを開始するトリガ プログラムの名前です。 トリガ プログラムは、システム パスで表わすか、または明示的に指定する必要があります。
サーバー ファイル名	Registry File Generatorで作成されるレジストリ ファイルの名前です。
クライアント ファイル名	分散サーバーを呼び出すことになるクライアント マシン上で使うために、Registry File Generator作成されるレジストリ ファイルの名前です。

3. サーバー情報について、以下のうちの 1 つを選択します。

簡易	標準のOLE オートメーション サーバー
プロセス内	OLEオートメーション サーバー
分散	サーバーおよびクライアントは、ネットワーク上の別々のマシンで実行されます。分散クライアントのレジストリ項目は、必ず REGEDIT4 形式で生成する必要があります。

OLE オートメーション サーバーを実行することになる、リモート サーバーのマシン名がわかっていれば、それを入力することができます。

それに代わる方法としては、アプリケーションおよびレジストリ ファイルを配布した後、Microsoft 提供プログラムである DCOMCNFG を使って、クラインアント マシンおよびサーバー マシンを設定することができます。

ファイルにある情報をレジストリに入力します。 コマンド ラインに以下のように入力します。

```
regedit  registryfile
```

パラメータ

registryfile	前述の手順 2 で作成したファイルの名前です。

24.5 OLE**オートメーション クライアントの作成方法**

OLEオートメーション クライアントを作成するには、次の手順を行います。

1. プログラムにチェッカー指令 OOCTRL(+P) を設定します。

```
$SET OOCTRL(+P)
```

このチェッカー指令を使って、ランタイム システムを COBOL データ タイプからOLEデータ タイプにマッピングすることができます。

2. 各OLEクラスをCLASS-CONTROL段落にあるObject COBOLプロキシ クラスにマップします。Object COBOL class-nameにOLEクラスを指定し、次の形式の文を使ってそれをOLE ドメインの中に指定します。

```
class-name IS CLASS "$OLE$windows-registry-name"
```

windows-registry-name は以下のうちの 1 つです。

- サーバーのProgID
- サーバーのCLSID

ProgID (プログラム ID)およびCLSID (クラス ID)は、Windows レジストリで定義されています。

3. オブジェクトCOBOL OLEクラスのインスタンスを作成します。

"new"メッセージを Object COBOLプロキシ クラスに送信します。 OLEサーバーがまだ実行されていない場合は、これにより開始されます。 OLEサーバーにメッセージを 送信するために使用できる、Object COBOL ハンドルが戻されます。

4. メッセージを OLE サーバーに送信します。

5. プロパティにアクセスするには、プロパティ名の前に、"Set" (プロパティを設定するとき) または"Get" (プロパティを取り出すとき) を付けます。

プロパティを設定するときは単一のパラメータを渡し、プロパティを取得するときは値が RETURNING パラメータとして返されます。

OLE オートメーション クライアントの例

この例では、Windows 版 Word を起動しOLE サーバとしてそれを使用する、OLEオートメーション クライアントの概略を示します。

```
$set ooctrl(+P) mfoo

 class-control.

 ...
```

- このプログラムに対し、Windows 版 Word を OLE オートメーション サーバーとして登録します。Windows 版 Word がインストールされると、word.basic という名前で Windows レジストリに入力されます。

```
   word is class "$OLE$word.basic"
   ...
   .
working-storage section.
```

- サーバーに参照を保持するためのデータ項目

```
01 wordServer       object reference.
 ...

 procedure division.
 ...
```

- Windows 版 Word を起動し、ハンドルを取得します。

```
 invoke Word "new" returning wordServer
...
```

- OLE サーバーにメッセージを送信します。

```
invoke wordServer "fileOpen" using aFilename
```

第25章 OLE コンテナのテンプレート

この章では、OLEコンテナのテンプレートについてと、GUIアプリケーションのテンプレートについて説明します。

25.1 概要

OLEコンテナ テンプレートは、GUIアプリケーション テンプレートの拡張で、基本的なGUI アプリケーションをOLE コンテナ アプリケーションに調整するような機能が追加されています。OLE コンテナ テンプレートを使用する前に GUI テンプレートに精通しておくことをお勧めします。GUI テンプレートの詳細については、25.3「GUIアプリケーションのテンプレート」を参照してください。

提供されているコンテナ テンプレートは NetExpress プロジェクトです。これを新しいディレクトリにコピーし、再度ビルドして実行すると、アプリケーションの作業を開始することができます。

コンテナ テンプレートを使ってアプリケーションを作成する方法については、25.2「OLEコンテナ アプリケーションの作成方法」を参照してください。

テンプレートをコピーして実行すると、次のようになります。

- 「File」、「Edit」と 「Help」の3つのサブメニューを持つメイン ウィンドウが表示されます。
- 「File」サブメニューには、「New」、「Open」、「Save」、および 「Exit」の項目が表示されます。「Exit 」をクリックすると、アプリケーションを終了します。「Open」をクリックすると、「File Open」ダイアログ ボックスが表示されます。 OLEドキュメント コンテナのファイル (拡張子 .odc) をオープンしたり、保存したりできるように、デフォルトのロジックが含まれています。
- 「Edit」サブメニューには、「Cut」、「Copy」、「Paste」、「Paste Special」、「Delete」、「Insert New Object」、および 「Links」の項目が表示されます。

  「Insert New Object」では、クライアント領域に OLE ドキュメントを挿入することができます。新規 OLE ドキュメントに挿入するときは、既存のファイルにあるドキュメントを挿入するか、またはスクラッチから新しいドキュメントを作成するかのいずれかが選択できます。挿入されているドキュメントをダブルクリックするとき、または新しいドキュメントを作成するとき、そのドキュメントに対して登録されているアプリケーションが起動します。アプリケーションによっては、OLE コンテナ内部で実行するもの(インプレース アクティベーション)と、外部で実行するもの(アウトサイド イン アクティベーション)があります。

  「Cut」、「Copy」および 「Paste」では、アプリケーションと Windowsのクリップボードの間で、OLEドキュメントを移動させることができます。作成したアプリケーションでこの動作を確認したい場合は、オブジェクトを挿入し、それを別のものにカット アンド ペーストします。「Paste Special」では、クリップボードの中身をOLE オブジェクトとしてでなく貼り付けることができます。

- 「Help」サブメニューには、「Help」と 「About」の項目が表示されます。「About」をクリックすると、アプリケーションのテンプレートについての情報と共にダイアログ ボックスが表示されます。「Help」をクリックすると、NetExpress のヘルプ システムが開始されます。

- ツールバーとステータス バーが表示されます。これについては、GUIアプリケーションのテンプレートで説明されています。

- ドラッグ アンド ドロップがサポートされています。作成したアプリケーションでこの動作を確認したい場合は、OLE コンテナ テンプレートの2つのコピーを開始し、その片方にオブジェクトを挿入して、それをもう片方にドラッグします。コピーの開始方法については、以下の「コピーの開始方法」を参照してください。

テンプレートは、アプリケーション作成の例として、またはスターティング ポイントとして動作するようデザインされています。多くのアプリケーションが、すでにある動作に新しい動作を追加するだけで作成できます。一部のアプリケーションでは、既存の動作を変更することも必要になります。

OLE コンテナのテンプレートを使用するためのステップごとの手順については、25.2「OLEコンテナ アプリケーションの作成方法」を参照してください。テンプレートの動作に関する特別なバックグラウンド、およびアプリケーションの作成方法のアドバイスについては、25.3「GUIアプリケーションのテンプレート」および第23章「OLEコンテナおよびサーバのサポート」を参照してください。

コピーの開始方法

コンテナ アプリケーションの特別なコピーを開始するには、次の作業を行います。

1. Windowsの「スタート」メニューにある NetExpressグループから、「NetExpress コマンド プロンプト」を起動します。

2. テンプレート アプリケーションのコピーを作成したディレクトリへ、ディレクトリを変更します。

3. ディレクトリをdebugサブディレクトリに変更します。

4. 次のコマンドを入力します。

   ```
   runw cntrtrig
   ```

25.2 OLEコンテナ アプリケーションの作成方法

テンプレートから OLE コンテナ アプリケーションを作成するには、次の手順を行います。

1. Windows の 「スタート」 メニューの 「プログラム - Micro Focus NetExpress 3.0 J - NetExpress コマンド プロンプト」 をクリックします。

2. 次のコマンドを入力します。

```
mfnetx /clone:c:¥netexpress¥base¥source¥template¥ole¥containr¥cntrtplt.app
```

ここで *c*は、ユーザのNetExpress ドライブのことです。

3. 「新規プロジェクトのクローン」ダイアログ ボックスで、ディレクトリ名とプロジェクト ファイル名を指定します。

4. 「OK」 ボタンをクリックして、アプリケーション ファイルをコピーします。

5. 「NetExpress」の「プロジェクト」メニューで、「すべてをリビルド」 をクリックします。

6. 「アニメート」メニューで「実行」 をクリックし、テンプレート コンテナの実行を確認します。

   この時点で、スケルトン OLE コンテナ アプリケーションが実行されます。それには、ツールバー、ステータス バー、メニューのヒント、およびツールの説明が含まれています。

7. OLE コンテナの「Edit」メニューで 「Insert New Object」 をクリックします。

   これにより、「Insert Object」ダイアログ ボックスがオープンします。 このダイアログ ボックスを使って、異なるタイプの OLE オブジェクトを、テンプレート アプリケーションのクライアント領域に挿入することができます。 このダイアログ ボックスの「オブジェクトの種類」リストの内容は、コンピュータにインストールされているソフトウェアにより異なります。

   タイプとして「ビットマップ　イメージ」が利用できることが前提となっています。これは、「ペイント」が標準の Windows インストールの一部となっているためです。 表示されない場合は、以下の「「ビットマップ　イメージ」が「オブジェクトの種類」リストに表示されない場合」を参照してください。

8. 「ビットマップ　イメージ」を選択して、「OK」ボタンをクリックします。

   これにより、Windows の 「ペイント」 プログラムが起動し、OLE コンテナのクライアント領域内に表示されます。「ペイント」 のツールバーとメニューが、OLE コンテナ アプリケーションに追加されます。 これはインプレース アクティベーションとして知られています。

9. 「ペイント」 を使って線や図形をいくつか描き、次にペイント枠の外側の領域をクリックします。

   「ペイント」 は終了しますが、描いたビットマップは残されており、枠で囲まれています。 ビットマップをダブルクリックすると、「ペイント」 が再起動し、プログラムを編集することができます。

10. 「Template Application」 ウィンドウの 「File」 メニューから 「Exit」 を選択します。

テンプレートに機能を追加し、完全な機能を持ったアプリケーションに変えられるようになりました。

「ビットマップ　イメージ」が「オブジェクトの種類」リストに表示されない場合

「ビットマップ　イメージ」が「オブジェクトの種類」リストに表示されない場合は、以下のようにします。

1. Windowsの「スタート」メニューの「コントロール パネル」から、「アプリケーションの追加と削除」ダイアログ ボックスを表示します。
2. 「Windows ファイル」タブをクリックします。
3. 「ファイルの種類」リストで 「アクセサリ」 を選択します。
4. 「詳細」ボタンをクリックします。
5. 「ファイルの種類」リストで「ペイント」をクリックして、チェックマーク付きで表示されるようにします。
6. 「OK」 ボタンをクリックします。
7. 「適用」ボタンをクリックします。
   Windows インストール ディスクがまだロードされていない場合は、プロンプトが表示されます。
8. この手順の前に行っていた手順に戻ります。

25.3 GUIアプリケーションのテンプレート

GUI アプリケーションテンプレートは、Object COBOL クラス ライブラリを使用してグラフィカル アプリケーションを最短時間で構築する上で便利なショートカット フレームワークです。 提供されているテンプレートは NetExpress プロジェクトです。これを新しいディレクトリにコピーし、再度ビルドして実行すると、アプリケーションの作業を開始することができます。

GUIテンプレートを使ってアプリケーションを作成する方法については、後述の「GUIテンプレートを使ったアプリケーションの作成方法」を参照してください。

25.3.1 概要

テンプレートをコピーして実行すると、次のようになります。

- 「File」と 「Help」の2つのサブメニューを持つメイン ウィンドウが表示されます。
- 「File」サブメニューには、「New」、「Open」、「Save」、および 「Exit」の項目が表示されます。 「Exit 」をクリックすると、アプリケーションを終了します。「Open」をクリックすると、「File Open」ダイアログ ボックスが表示されます。
  ファイルをオープンしようとすると、ファイルをオープンすることができないことを示すメッセージ ボックスが表示されます。これは、ファイルをオープンするためのロジックが実際には存在していないためで

す (このロジックの作成は、アプリケーション作成作業の一部に含まれます)。 また、これはオープンしようとするファイルのタイプによって異なります。

- 「Help」サブメニューには、「Help」と 「About」の項目が表示されます。「About」をクリックすると、アプリケーションのテンプレートについての情報と共にダイアログ ボックスが表示されます。「Help」をクリックすると、NetExpress のヘルプ システムが開始されます。

- ウィンドウの一番上に沿った横方向のツールバーには 「New」、「Open」、「Save」 アイコンが表示されます。ツールヒント（ボタンをクリックすると起こる動作を示す小さなテキスト ラベル）を表示するためには、ツールバーのボタンの上にマウス ポインタを移動します。これらのアイコンは、「File」サブメニューのメニュー項目に対応しています。

- ウィンドウの一番下に沿って横方向にあるステータス バーは 4 つのセクションに分割されます。メニュー ヒントが最初のセクションに表示され、INS、CAPS、NUM のステータス キー インジケータが次の 3 つのセクションに表示されます。

テンプレートは、アプリケーション作成の例として、またはスターティング ポイントとして動作するようデザインされています。多くのアプリケーションが、すでにある動作に新しい動作を追加するだけで作成できます。一部のアプリケーションでは、既存の動作を変更することも必要になります。

テンプレートは、Object COBOL のオブジェクト指向の機能を使用して書かれ、完全な OO ベースのアプリケーションを設計し、作成したい場合にできるように構成されます。しかし、手続き COBOL を使用してアプリケーションを作成したい場合、そのように行うこともできます。

このヘルプは、作業を始める前にテンプレートの基本設計に関する説明を読むか、すぐに固有のアプリケーションに変更を行い、構築しはじめることができるように構成されています。

GUIアプリケーションのテンプレートを使用するためのステップごとの手順については、次の各項目を参照してください。

- 新規の出力シグナルの定義方法
- ステップ 1：アプリケーション メニューの変更方法
- ステップ 2：ボタンとセパレータをツールバーに追加する
- ステップ 2：ツールバー ビットマップの変更方法
- ステップ 3：ダイアログの追加方法
- ステップ 4：ダイアログ ボックスの制御方法
- ステップ 5：イベントで起動されるメソッドの作成方法
- ステップ 6：情報のクライアント領域への設定方法

- ステップ 7：シグナルの処理方法
- ステップ 2：シグナルの送信方法

テンプレートの動作に関する特別なバックグラウンド、およびアプリケーションの作成方法のアドバイスについては、25.3.2「アプリケーション テンプレート構造」および25.3.3「アプリケーションの構築」を参照してください。

GUIテンプレートを使ったアプリケーションの作成方法

GUIテンプレートを使ってアプリケーションを作成するには、次の手順を行います。

1. Windows の「スタート」メニューから 「プログラム - Micro Focus NetExpress 3.0 J- NetExpress コマンド プロンプト」 をクリックします。
2. 次のコマンドを入力します。

```
mfnetx /clone:c:¥netexpress¥base¥source¥template¥guiapp¥tpltgui.app
```

   ここで *c* は、ユーザのNetExpress ドライブです。

3. 「新規プロジェクトのクローン」ダイアログ ボックスで、ディレクトリ名とプロジェクト ファイル名を指定します。
4. 「OK」 をクリックして、アプリケーション ファイルをコピーします。
5. 「NetExpress」の「プロジェクト」メニューで、「すべてをリビルド」 をクリックします。
6. 「アニメート」メニューで「実行」 をクリックし、テンプレート GUI アプリケーションの実行を確認します。

   この時点で、スケルトン アプリケーションが実行されます。それには、ツールバー、ステータス バー、メニューのヒント、およびツールの説明が含まれています。 s
7. 「Template Application」 ウィンドウの 「File」 メニューから 「Exit」 を選択します。

テンプレートに機能を追加し、完全な機能を持ったアプリケーションに変えられるようになりました。

25.3.2 アプリケーション テンプレート構造

GUI アプリケーション テンプレートは、ビジネスとインタフェース ロジックを 2 つの別のコンポーネントに分割します。分割する場合は、特に次のような利点があります。

- インタフェースとビジネス ロジックは別の個人またはチームにより作成することができます。

- ビジネス ロジックに影響を与えずに、インタフェースの設計を変更することができます。

2 つのコンポーネントは次のとおりです。

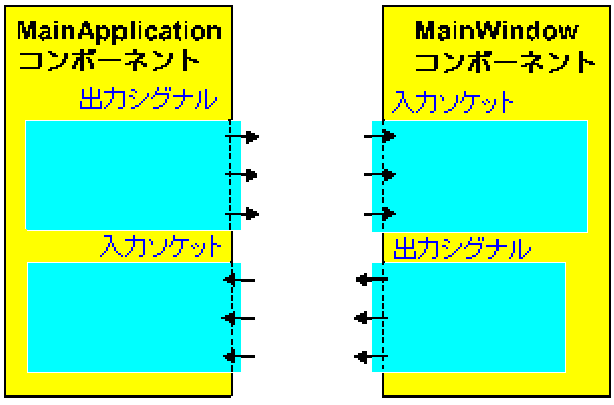
- MainWindow コンポーネント

  メイン アプリケーション ウィンドウの表示形式を定義し、グラフィカル インタフェースをコントロールします。

- MainApplication コンポーネント

  アプリケーションのビジネス ロジックを表します。単純なアプリケーションでは、MainApplication はすべてのビジネス ロジックを含むことができます。より複雑なアプリケーションでは、ビジネス ロジックは多くの異なるプログラムまたは Object COBOL クラスを含みます。この場合、MainApplication はインタフェースとビジネス ロジックをリンクします。

2 つのコンポーネントの間の通信はシグナルによります。



25.3.3 アプリケーションの構築

グラフィカル インタフェースを使用してアプリケーションを作成するとき、埋め込まれた多くのコードを作成します。各コードはアプリケーションのエンド ユーザにより選択されたアクションに対応しています。GUI アプリケーションが開始され、初期化されると、ほとんどの時間はエンドユーザによりアクションに対応するのを待つことに費やされます。

エンドユーザのアクションは、イベントとしてインタフェースに渡されます。インタフェースとビジネス ロジックは、シグナルを通じて通信します。

次の表は一般的なアプリケーションのシーケンス、アプリケーションのエンドユーザとMainWindow コンポーネント（インタフェース ロジック）および MainApplication コンポーネント（ビジネス ロジック）の間のコントロールのフローを示します。

エンドユーザ	MainWindow	MainApplication
	イベントを待ちます	
「Open」アイコンをクリックします	Open イベントを受け取ります	
	「Open」 ダイアログ ボックスを表示します	
	イベントを待ちます	
ファイルを選択し、「OK」 をクリックします	Dialog-OK イベントを受け取ります	
	Open-file シグナルを送信します	Open File シグナルを受け取ります
		ファイルを呼び出します
		display file シグナルを送信します
	display file シグナルを受け取ります	
	ファイルを表示します	
	イベントを待ちます	

GUI テンプレートは、インタフェースとビジネス ロジックを別の個人やグループが作成することができるように設計されています。ただし、MainWindow と MainApplication の間で受け渡されるシグナルとデータに関してあらかじめ合意していることが前提となります。

次の項では、手順ごとの説明があります。

- 25.3.4 シグナルの定義
- 25.3.5 ユーザ インタフェースの作成
- 25.3.6 ビジネス ロジックの作成

25.3.4 シグナルの定義

GUI アプリケーションを作成するための最初の手順は、使用中のアプリケーションのビジネス ロジック（MainApplication）とインタフェース ロジック (MainWindow)の間で受け渡されるすべてのシグナルを定義することです。インタフェースにより送信されるシグナルの一つのセットとビジネス ロジックで送信される一つのセットのシグナルを定義する必要があります。

一部のシグナルは、SymbolTables を使用してデータを搬送します。したがって、アプリケーション コンポーネント

の間で受け渡したい異なるすべてのデータにシンボリック名を定義することも必要です。

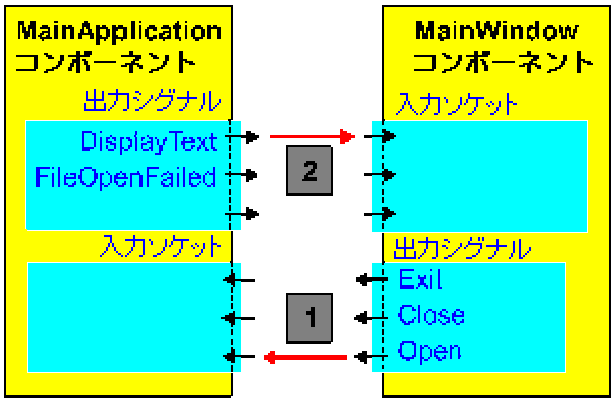
シグナルとともに送信するデータのタイプは通常、CharacterArray または OrderedCollection オブジェクトのいずれかです。OrderedCollectionは、一つのシグナルをもつ複数のCharacterArrayを関連付ける方法を与えます。

たとえば、ファイルをオープンし、ウィンドウにそれを表示する単純なアプリケーションを書きたいことがあります。次のシグナルおよびシンボルを定義することができます。

シグナル送信元	シグナル	シンボル	データの説明
MainWindow	open-signal	filename-sym filepath-sym	ファイルの名前 ファイルのパス
MainApplication	displayText-signal	textlines-sym	表示用のCharacterArrayのOrderedCollection
MainApplication	fileOpenFailed-signal	filename-sym filepath-sym	ファイルの名前 ファイルのパス

ファイル名とパスは、固有の名前付きシンボルを使用して一つの文字列で表すことができます。しかし、ファイルのテキストのすべての行に対して名前付きシンボルを定義することができないため、別のCharacterArray としてテキストの各行を OrderedCollection に入れ、それを送信します。OrderedCollection は、同種のデータ要素をグループ化するための便利な方法であり、ListBox などのグラフィック コントロールは一つのメッセージのみを使用してコレクションのすべての要素を表示します。

次の図は、MainWindow と MainApplication がシグナルを送信していることを示します。詳細な説明については、以下に番号付きのラベルの説明がありますので参照してください。



1. MainWindow コンポーネントは open シグナルを送信します。ファイル名とパスは、SymbolTable のデータとして搬送されます。

2. MainApplication コンポーネントの場合、displayText シグナルを送信します。データは OrderedCollection 内の CharacterArray のセットとして搬送されます。

新規の出力シグナルの定義方法

シグナルの定義のための手順を次に示します。

- ステップ1: 新規のシグナル名をシグナルのコピーファイルに追加します。
- ステップ2: シグナルに必要なシンボルを、シンボルのコピーファイルに追加します。

  シグナルが 1 つまたはそれ以上のデータ項目を繰り上げしている場合、識別元のシンボルをそれぞれに 1 つずつ定義する必要があります。
- ステップ3:"defineSignals"メソッドを更新します。

ステップ 1：新規のシグナル名をシグナルのコピーファイルに追加する方法

1. シグナルの追加先クラスの、シグナルのコピーファイルをオープンします。

   MainWindowクラスで送信されたシグナルは、tpltwin.cpy にあります。MainApplicationクラスで送信されたシグナルは、tpltapp.cpy で定義されます。
2. レベル 78 で、ヌルで終わるシンボルの定数を新規で追加します。

   78. *signal-name* z"*signalvalue*".

   パラメータ

   signal-name シグナルの名前です。

   signal-value 31 文字以内の長さの定数で、コピーファイル内でこれと重複するものがあってははなりません。

シグナルのコピーファイルの例については、19.5「シグナルの送信」の「シグナルのコピーファイル例」を参照してください。

ステップ 2：新規シンボルをシンボルのコピーファイルへ追加する方法

1. シンボルのコピーファイルをオープンします。デフォルトで、これは tpltkeys.cpy です。
2. レベル 78 で、ヌルで終わるシンボルの定数を新規で追加します。

   78. *symbol* z"*symboltext*".

   パラメータ

   symbol シグナルの名前です。

   symboltext 31 文字以内の長さの定数で、コピーファイル内でこれと重複するのもがあってはな

りません。

ステップ 3："defineSignals" メソッドの更新方法

1. ブラウザ ウィンドウを使って、シグナルをインタフェースに送信している場合は、MainWindowクラスの"defineSignals"メソッドを配置します。シグナルをビジネス ロジックに追加している場合は、MainApplicationクラスの"defineSignals"メソッドを配置します。

2. シグナル定義文を以下のようにして追加します。

   ```
   define signal signalname
   ```

   パラメータ

   signalname　　シグナルに定義された、レベル 78 の名前です。

25.3.5 ユーザ インタフェースの作成

イベントを作成する際にアプリケーションのエンドユーザはインタフェースと対話するため、アプリケーションは応答する必要があります。インタフェースに応答させたい各イベントに対して、通常はアプリケーション テンプレートの MainWindow でメソッドを作成することが必要です。

イベントの送信元は次の3つのいずれかです。

- メニュー
  エンドユーザがサブメニューの項目をクリックするとき。ツールボタンはメニュー項目にすべてマップされるため、ツールバー ボタンをクリックするとメニュー項目と同じタイプのイベントを生成します。

- ダイアログ ボックス
  エンド ユーザがプッシュボタンを押すか、リストの項目をダブルクリックするとき。リスト ボックスの任意のコントロールはイベントを生成できますが、これらはユーザが最も関心のあるものです。

- クライアント領域
  クライアント領域のイベントは、クライアント領域に表示されるものによって異なります。しかし、アプリケーションによっては、右クリックで選択したり、コンテキスト メニューを表示したり、マウス ボタンを下に押し下げて、ドラッグ操作を使用可能にすることができます。

インタフェース作成のための主な手順を下に示します。順番には正確に従う必要はありませんが、作業開始のための便利なガイドです。

- ステップ 1： メニューを作成し、各メニュー項目をメソッドにマップします。これらのほとんどのメソッドではダイアログ ボックスが表示されますが（下のステップ 4 を参照）、クライアント領域の表示オプ

ションを変更するか、シグナルを送信します（たとえば、現在のデータを保存するため）。
後述の「ステップ 1：アプリケーション メニューの変更方法」を参照してください。

- ステップ 2： ツールバーをデザインし、その上のボタンをメニュー項目にマップします。
後述の「ステップ 2：ボタンとセパレータをツールバーに追加する方法」、「ステップ 2：ツールバー ビットマップの変更方法」を参照してください。

- ステップ 3： Dialog Editor を使用してダイアログ ボックスをペイントしてから、指定されたテンプレートを使用して各ダイアログ ボックスのクラスを作成します。各ダイアログ ボックスのクラスには、MainWindow がエンドユーザによるデータ入力を取得し、表示のためにダイアログにデータを送信できるメソッドがあります。ダイアログ ボックス クラスは、MainWindow クラスが指定した論理イベントにユーザ イベント（プッシュボタンのクリックなど）もマップします。
後述の「ステップ 3：ダイアログ ボックスの追加方法」を参照してください。

- ステップ 4： 各ダイアログ ボックスを作成、表示するため、およびダイアログ ボックス イベントをメソッドにマップするために、MainWindowへメソッドを追加します。
後述の「ステップ 4：ダイアログ ボックスの制御方法」を参照してください。

- ステップ 5： イベントにより呼び出されるメソッドを作成します。メソッドはダイアログ ボックスを表示するか、キャンセルするか、アプリケーションにデータを渡すか、それからのデータを要求してシグナルを送信します。
後述の「ステップ 5：イベントで起動されるメソッドの作成方法」を参照してください。

- ステップ 6： クライアント領域を指定します。クライアント領域に対してリストボックスや複数行のエントリ フィールド (MLE) などのコントロールを使用することができます。これらにテキストを表示することが非常に簡単だからです。代わりにサブペインを使用したいため、クライアント領域に直接ペイントすることができます。ペイントは最大の柔軟性とコントロールを与えますが、作業量は増えます。
後述の「ステップ 6：情報のクライアント領域への設定方法」を参照してください。

- ステップ 7： 入力シグナルを処理するためのコードを作成します。ほとんどのシグナルはインタフェースからの要求が成功したか、失敗したかを受諾するものです。一部のシグナルはアプリケーションのエンドユーザに対して表示するためのデータも搬送します。
後述の「ステップ 7：シグナルの処理方法」を参照してください。

ステップ 1：アプリケーション メニューの変更方法

次の手順を参照してください。

- 手順1： メニュー定義を変更します(手順 1：メニュー定義の変更方法参照)。

- 手順2： メニューのヒントを追加します(手順 2：メニューのヒントの追加方法参照)。

- 手順3： シンボル ファイルを更新します(手順 3：シンボル ファイルの更新方法参照)。

- 手順4： メニュー マップを更新します(手順 4：メニュー マップの更新方法参照)。
- 手順5： 新規メニュー項目を選択したときに実行されるメソッドを追加します(手順 5：メニュー メソッドの作成方法参照) 。

手順 1：メニュー定義の変更方法

1. メニュー リソース ファイル tpltmenu.mnuをオープンします。
    a. 「プロジェクト」ウィンドウのリソース tpltres.rc をダブルクリックします。
    b. リソース ファイルのメニュー リソースをクリックします。
    c. 「tpltmenu.mnu」 をダブルクリックします。
2. メニュー定義を捜します。キーワード MENU で始まり、BEGIN と END キーワードの間に表示されます。
3. メイン メニュー定義にサブメニューを追加するか、別のサブメニュー定義の中に追加することができます。

    新規サブメニューを追加するためには、下記のとおり追加します。

```
POPUP submenu
BEGIN
menu-items
END
```

パラメータ

submenu	サブメニューに表示されるテキスト。
menu-items	下のとおり定義されます。

4. サブメニューにメニュー項目を追加するためには、メニュー項目を追加したいサブメニュー定義を検索し、下のとおり追加します。

```
MENU-ITEM menu-text symbol
```

パラメータ

menu-text	サブメニューに表示されるテキスト。

symbol　　コードの中からメニュー項目を参照する名前。識別子の中では、アンダスコア (_)を使用できますが、ハイフン(-)は使用できません。

5. メニュー ファイルに変更を保存します。

メニュー定義の例

次のコードは、GUI テンプレート アプリケーションのメニューを定義する.rcファイルの一部です。右下のIDシンボルは、COBOL統合開発環境のリソース エディタにより、レベル-78 データ項目としてCOBOL.cpyファイルに自動的に書きこまれます。

```
ID_MENUBAR MENU
BEGIN
   POPUP "&File"
   BEGIN
      MENUITEM "&New",                    ID_FILE_NEW
      MENUITEM "&Open",                   ID_FILE_OPEN
      MENUITEM "&Save",                   ID_FILE_SAVE
      MENUITEM SEPARATOR
      MENUITEM "E&xit",                   ID_APP_EXIT
   END
   POPUP "&Help"
   BEGIN
      MENUITEM "&Help",                   ID_HELP_HELP
      MENUITEM "&About",                  ID_HELP_ABOUT
   END
END
```

手順 2：メニューのヒントの追加方法

メニュー ヒントは任意選択なので、必要ない場合は手順 3 に進んでください。

1. tpltmenu.mnu で定義したストリングテーブル定義を検索します（前のトピックで開いています）。ストリングテーブル定義は、キーワード STRINGTABLE で始まります。ストリングは BEGIN と END キーワードの間で定義されます。

2. 希望のそれぞれのメニュー ヒントに対して新しい文字列を追加します。ツール ヒントを定義するために、任意に同じ文字列を使用することができます。

   新規文字列を追加するためには、下記のとおり追加します。

   symbol "menu-hint/n*tool-tip"*

   パラメータ

symbol	関連付けられたメニュー項目の識別子
menu-hint	ヒント テキスト
tool-tip	ツール ヒント

3. 変更を保存するために 「ファイル」 メニューの 「保存」 をクリックして、編集ウィンドウを閉じます。

メニュー ヒント ストリングテーブルの例

次のストリングテーブルは 3 つのメニュー項目、ツール ヒントおよびメニュー項目と関連つけられたツールバーボタンのメニュー ヒントを定義します。

```
STRINGTABLE DISCARDABLE
BEGIN
    ID_FILE_NEW           "Create a new document¥nNew"
    ID_FILE_OPEN          "Open an existing document¥nOpen"
    ID_FILE_SAVE          "Save the active document¥nSave"
END
```

手順 3：シンボル ファイルの更新方法

1. Symbol Editor を起動します（リソース メニューのシンボル をクリックします）。

2. 「新規」 をクリックします。
3. 「名前」フィールドに新規シンボルの名前を入力します。
4. 「値」フィールドに数値 id を入力します（各シンボルには別々の id を指定する必要があります）。
5. 「適用」 をクリックします。

   Symbol Editor は数値 id が別のシンボルにしようされた id と重複している場合にはエラー メッセージを表示します。
6. メニュー ファイルに新しいシンボルを追加するたびに、ステップ 2 から 5 までを繰り返します。
7. Symbol Editor を終了する場合は、「OK」 をクリックします。

手順 4：メニュー マップの更新方法

1. アプリケーションの MainWindow で"defineMenuMap"メソッドを検索します。

   そのメソッドをすばやく見つけるためにはブラウザを使用してください。
2. エンドユーザがメニュー項目を選択するたびに実行する MainWindow クラスのメソッドにメニュー項目の識別子をリンクしながら、それぞれの新規メニュー項目に対してマップ文を追加します。

   ```
   MAP MENUID identifier AT resource
   TO method
   ```

   パラメータ

identifier	メニュー項目の rc ファイルで定義された名前。任意のアンダスコア (_) は Object COBOL プログラムで使用するためにハイフン (-) に変更されます。
resource	このメニューのリソース モジュール。デフォルトではデータ項目 osResource にあります。
method	メソッドを指定するヌルで終了したリテラルの名前指定。GUI テンプレートで使用されている命名規則にしたがうために、メソッドに名前をつけます。 "onEvent*MenuItem*"

メニュー マップの例

次のメニュー マップは GUI テンプレートにより指定されたデフォルトのマッピングに加えて、特定のアプリケー

ションに追加された追加のマッピングを示します（USER-CODE と END-USER コメントの間）。

```
method-id. "defineMenuMap".
procedure division.
```

- ---**デフォルト メニュー項目のマッピング**

```
   map menuId ID-HELP-HELP at osResource to z"onEventHelp"
map menuId ID-HELP-ABOUT at osResource to z"onEventAbout"
map menuId ID-APP-EXIT at osResource to z"onEventExit"
```

- ---USER-CODE.**この特定のアプリケーションのメニュー項目のマッピング**

```
   map menuId ID-ACCOUNT-NEW at osResource to z"onEventNewAcc"
map menuId ID-ACCOUNT-QUERY at osResource to z"onEventQuery"
map menuId ID-ACCOUNT-GET at osResource to z"onEventGet"
```

- ---END-USER.

```
   exit method.
end method "defineMenuMap".
```

手順 5：メニュー メソッドの作成方法

メニュー マップで定義した名前と一致する名前を衣装して、MainWindow クラスにメソッドを追加します。メソッドのコードは、メニュー項目が選択されるたびに実行されます。

メニュー メソッドはダイアログ ボックスを表示するか、アプリケーション コンポーネントからデータを要求する

か、インタフェースの状態だけを変更します。

メニュー メソッドの例

次のメニューは GUI テンプレート MainWindowクラスからのものであり、アプリケーションのエンドユーザが「File」 メニューの 「Open」 をクリックすると呼び出されます。

```
method-id. "onEventOpen".


local-storage section.
01 lsDataBlock            object reference.
01 lsOpenDialog           object reference.
01 lsString               object reference.
01 lsMessage              object reference.
procedure division.
```

- ---「File Open」ダイアログが表示されます。

```
   invoke OpenDialog "new" using self
returning lsOpenDialog
invoke lsOpenDialog "show"
```

- ---「File Open」ダイアログが最終処理されます。

- ---ダイアログから情報を検索します

```
   invoke lsOpenDialog "getFile" returning lsString
if lsString <> null
```

```
invoke SymbolTable "new" returning lsDataBlock
put value lsString into table lsDataBlock
at symbol filepath-key
invoke lsOpenDialog "getFileTitle" returning lsString
put value lsString table lsDataBlock symbol filename-key
```

- -------**オープンするシグナルが送信されます。**

```
invoke self "signal" using open-signal lsDataBlock
```

- -------**クリーンアップされます。**

```
invoke lsDataBlock "deepFinalize" returning lsDataBlock
end-if
invoke lsOpenDialog "finalize" returning lsOpenDialog
exit method.
end method "onEventOpen".
```

関連項目

- ステップ 3：ダイアログ ボックスの追加方法
- ステップ 4：ダイアログ ボックスの制御方法
- ステップ 7：シグナルの処理方法
- ステップ 5：イベントで起動されるメソッドの作成方法
- ステップ 1：アプリケーション メニューの変更方法

- ステップ 2：シグナルの送信方法
- 新規の出力シグナルの定義方法
- ステップ 2：ボタンとセパレータをツールバーに追加する方法
- ステップ 2：ツールバー ビットマップの変更方法
- ステップ 6：情報のクライアント領域への設定方法

ステップ 2：ボタンとセパレータをツールバーに追加する方法

1. MainWindowクラス（tpltwin.cbl）で、"defineToolbarMap" メソッドを検索します。
2. ツールバー ボタンを追加するには、以下の文を追加します。

```
map button index at osToolbar to
    menuId symbol at osResource
```

パラメータ

index	整数値です。 ボタンの位置をツールバー ビットマップに示します。
symbol	メニュー項目のシンボルです。各ツールバー ボタンをメニュー項目にマップします。エント ユーザがツールバー ボタンをクリックすると、そのメニュー項目を選択する場合と同様の効果が得られます。ボタンを使って、メニュー項目に定義されたメニューのヒントとツールバーの説明を選択することもできます。

3. セパレータを追加するには、以下の文を追加します。

```
invoke osToolbar "addSeparator"
                 returning button
```

ツールバー ボタンおよびセパレータが、追加した順序で表示されます。

「ステップ 2：ツールバー ビットマップの変更方法」の項も参照してください。

ツールバー マップの例

下のメソッドは、GUI アプリケーション テンプレート MainWindow クラスに基づいたウィンドウ ベースのツール

バー マップを定義します。追加される最初の 3 つのボタンは、デフォルトのボタンです。その後で、ツールバーにセパレータを置くためにコードが追加されます。その後にツールバー ビットマップで定義された 6 番目のボタンが続きます。指定されたデフォルトのツールバー ビットマップ(tplttool.bmp)を使用している場合、これはヘルプ ボタンです。

```
method-id. "defineToolbarMap".
local-storage section.
01 lsId              pic x(4) comp-5.
01 lsIndex           pic x(4) comp-5.
01 lsTemp            object reference.
procedure division.
```

- ---USER-CODE. **デフォルトのツール ボタンを変更するためには、このコードを変更してください。**

- ---**ツールバーに 「**New File**」 ボタンを追加します**

```
   map button 1 at osToolbar to menuId ID-FILE-NEW
at osResource
```

- ---**ツールバーに 「**Open File**」 ボタンを追加します**

```
   map button 2 at osToolbar to menuId ID-ACCOUNT-QUERY
at osResource
```

- ---**ツールバーに 「**Save File**」 ボタンを追加します**

```
    map button 3 at osToolbar to menuId ID-ACCOUNT-GET
at osResource
```

- ---END USER.

- ---USER-CODE. **ここに後で使用するツールバー ボタンを追加します**

```
    invoke osToolbar "addSeparator" returning lsTemp
map button 6 at osToolbar to menuId ID-HELP-HELP
at osResource
```

- END-USER.

```
    exit method.

end method "defineToolbarMap".
```

ステップ 2：ツールバー ビットマップの変更方法

1. プロジェクト ウィンドウで tpltres.rc をダブルクリックして、リソース マネージャを開始します。
2. リソース マネージャでビットマップをダブルクリックして、これを拡張します。
3. ID_TPLTTOOL をダブルクリックして、これをビットマップ エディタにロードします。
4. ビットマップを編集して、必要なボタンを描画します。

ツールバー上の各ボタンは、16 x 16 ピクセルです。Microsoft Image Editor（デフォルトのエディタで、Microsoft Windows SDK の一部として提供されています）を使用中の場合は、左上のピクセルの位置が 0,0 です。2 番目のボ

タンの開始位置は 16,0 、3 番目は 32,0、というようになります。

デフォルトのビットマップの長さは、ボタン 9 つ分です。これでは小さ過ぎる場合は、新規のビットマップを作成して tpltool.bmp として保存します。

「ステップ 2：ボタンとセパレータをツールバーに追加する方法」の項も参照してください。

ステップ 3：ダイアログ ボックスの追加方法

次の手順は、GUI アプリケーション テンプレートで使用するダイアログ ボックス クラスを作成する方法を示します。

- 手順1： 新規ダイアログ ボックスをペイントします(手順 1：新規ダイアログ ボックスのペイント方法参照)。
- 手順2： 新規ダイアログ ボックスを表すクラスを作成します(手順 2：ダイアログ ボックスのクラスの作成方法参照)。

  別の Object COBOL クラスにより各ダイアログ ボックスを表すことにより、MainWindow クラスのコードを簡素化します。
- 手順3： コントロールにアクセスするためのコードを追加します(手順 3：コントロールにアクセスするためのコードの追加方法参照)。

  問い合わせたり、状態を変更したり、イベントを受け取りたいコントロールのオブジェクト ハンドルを取得する必要があります。
- 手順4： イベントを処理するためのコードを追加します(手順 4：イベントを処理するためのコードの追加方法参照)。

  「OK」 と 「Cancel」 を処理するデフォルト コードはすでに指定されています。
- 手順5： ダイアログ ボックスの情報を設定し、取得するためのコードを追加します(手順 5：コントロールからのデータの設定と取得方法参照)。
- 手順6： ダイアログ ボックスが表示されたときに特定のコントロールにフォーカスを設定します(手順 6：特定のコントロールにフォーカスをセットする方法（オプション）参照)。

たとえば、エントリ フィールドにフォーカスを設定する場合、ダイアログ ボックスが表示されるとき、テキスト カーソルがユーザが入力を開始するためのエントリ フィールドに表示されます。

手順 1：新規ダイアログ ボックスのペイント方法

1. リソース エディタに読み込むために、プロジェクト ファイル リストのデフォルト(tpltres.rc)のアプリケー

ション リソース ファイルをダブルクリックします。

2. Dialog Editor をオープンするために、リソース ファイル ビューでダイアログ ボックス グループをダブルクリックします。

3. 「編集」メニューの 「新規ダイアログ」 をクリックして、新規ダイアログ ボックスを作成します。新しいダイアログ ボックスに固有の名前を指定します。Object COBOL コードの中でダイアログ ボックスを作成する場合にこれが必要です。

4. ダイアログ ボックスにラベルとガジェットを追加します。ダイアログ ボックスに 「OK」 と 「キャンセル」 ボタンがある場合、シンボル IDOK と IDCANCEL にそれらをマップします。

5. それらの変更を保存し、Dialog Box エディタを終了します。

手順 2：ダイアログ ボックスのクラスの作成方法

1. tpltdbox.cbl を新規ファイルにコピーします。これは、最初のテンプレートのダイアログ ボックス クラスです。

2. プロジェクトに新規ファイルを追加し、それを開いて編集します。

3. プログラムの一番上の Class-ID 見出しを編集します。

```
Class-id. classname

data is private

inherits from superclass
```

パラメータ

classname	ソース コードのクラスを識別する名前。
superclass	Modal またはModelessのいずれか。ダイアログ ボックスがアプリケーション モーダルまたはモードレスのいずれであるかを決めます。

4. Class-Control 段落を変更して、このクラスのクラス名とファイル名を入れます。行を次のとおり変更します。

```
DialogBoxTemplate is class "tpltdbox"
```

を以下のとおり変更します。

```
classname is class "filename"
```

パラメータ

classname 上記の手順 2-2 のクラスに指定した名前。

filename 上記の手順 2-1 でコピーした宛先のファイル名。

5. ソースコードの最後の行にある、End Class 見出しを変更します。

```
end class classname
```

パラメータ

classname 上記の手順 2-2 のクラスに指定した名前。

この段階でダイアログ ボックスのクラス コードのアウトラインができますが、固有の動作はありません。

手順 3：コントロールにアクセスするためのコードの追加方法

1. ダイアログ ボックス クラスの中のオブジェクト記憶域セクションを検索します。
2. アクセスしたい各コントロールに対して Object Reference タイプのデータ項目を追加します。
3. ダイアログ ボックス クラスの中の"create"メソッドを捜します。
4. 各ガジェットのハンドルを取得するために、メッセージ「getObjectFromId」を送信します。

```
move identifier to lsId
invoke self "getObjectFromId"
using lsId
returning osObject
```

パラメータ

identifier ガジェットの DialogEditor で定義したシンボル。任意のアンダスコア (_) は Object COBOL プログラムで使用するためにハイフン (-) に変更されます。

lsId PIC X(4) COMP-5.

osObject OBJECT REFERENCE.

他のメソッドでそれにアクセスする必要がある場合、オブジェクト記憶域セクションでこれを宣言します（たとえば、入力フィールド データを設定または取得するためです）。このメソッドでそれにアクセスすることだけが必要な場合、ローカル記憶域セクションでこれを宣言します（たとえば、プッシュボタンのイベント ハンドリングをセットアップするためです）。

これらのデータ項目を使用して、状態を変更したり、データを呼び出しながらガジェットにメッセージを送信します。

ダイアログ ボックスのアクセス コントロールの例

このコードは、GUI アプリケーション テンプレートのダイアログ ボックス作成メソッドの一部を示します。指定されたコードは、シンボル EF-NUMBER を識別してコントロールのオブジェクト ハンドルを取得し、それをosNumberFieldデータ項目に入れます。

```
method-id. "create".

local-storage section.

01 lsResourceId              pic x(4) comp-5.

01 lsPushbutton              object reference.

01 lsParent                  object reference.

01 lsEvent                   pic x(4) comp-5.

01 lsLogicalId               pic x(4) comp-5.


procedure division.

invoke super "create"
```

- ---USER-CODE.OK & CANCEL **のデフォルト イベント ハンドリングをセットアップします。**

- **ダイアログに** OK **または** CANCEL **がない場合には、このコードを変更します。**

- pushbuttons.

...

- ---END-USER.

- ---USER-CODE.**他のガゼット/コントロールのハンドルを取得するためにコードを追加します**

```
    move EF-NUMBER to lsResourceId
invoke self "getObjectFromId" using lsResourceId
returning osNumberField
...
```

- ---END-USER.

```
   exit method.
end method "create".
```

- END-USER.

手順 4：イベントを処理するためのコードの追加方法

1. 必要な新しい論理イベントを定義します。

   ダイアログ ボックスの論理的イベントは、コピーファイルの数値を使ってレベル-78 データ項目として定義されます (デフォルトではtpltdial.cpy)。数値は 10000 より大きくすることが必要です。「OK」と「Cancel」

のイベントはデフォルトでは、10001 と 10002 として定義されています。

後述の「論理イベント コピーファイル例」を参照してください。

2. ダイアログ ボックス クラスの中の"create"メソッドを捜します。

3. "getObjectFromId"メッセージを使用してプッシュボタンまたはその他のコントロールを呼び出すためのコードを追加します。

   デフォルトの"create"メソッドにはすでに、プッシュボタン IDOK と IDCANCEL を処理するためのコードがあります。ダイアログ ボックスでこれらのシンボルに対して宣言されたプッシュボタンがない場合には、このコードを削除してください。

4. 各プッシュボタンについては、次のとおり追加します。

```
map event p2ce-clicked upon pushbutton
 to logical event logicalId upon self
```

パラメータ

logicalId	手順 4-1 で定義されたレベル - 78.
pushbutton	OBJECT REFERENCE. 手順4-3 で呼び出されたプッシュボタンに対するオブジェクト ハンドル。

エンドユーザがプッシュボタンをクリックすると、ダイアログ ボックスが論理イベントを発生させます。
MainWindow クラスの中のコードは論理イベントに応答でき、適切なアクションを実行します。

後述の「ダイアログ ボックス論理イベントの例」を参照してください。

論理イベントのコピーファイル例

次のコピーファイルは、新しい論理的イベント、dialog-apply を示します。

- ----------
- **ダイアログ ボックスの包括的論理イベント**
-
- Copyright (c) Micro Focus Ltd 1995.

- ----------

```
78 dialog-OK                value 10001.
78 dialog-Cancel             value 10002.
```

- USER-DATA.**次の新規イベントを追加します。**

```
78 dialog-apply              value 10005.
```

- END-USER.

ダイアログ ボックスの論理的イベントの例

このコードは、GUI アプリケーション テンプレートのダイアログ ボックス作成メソッドの一部を示します。示されたコードはシンボル BTN-APPLY でプッシュボタンをセットアップして、エンドユーザがクリックするときに論理的イベント DIALOG-APPLY を発生させます。

```
method-id. "create".
local-storage section.
01 lsResourceId              pic x(4) comp-5.
01 lsPushbutton              object reference.
01 lsParent                  object reference.
01 lsEvent                   pic x(4) comp-5.
01 lsLogicalId               pic x(4) comp-5.

procedure division.
```

```
invoke super "create"
```

- USER-CODE.OK & CANCEL **のデフォルト イベント ハンドリングをセットアップします。**
- **ダイアログに** OK **または** CANCEL **がない場合には、このコードを変更します。**
- pushbuttons.

```
 ...
```

- ---END-USER.
- ---USER-CODE. **他のガゼット/コントロールのハンドルを取得するためにコードを追加します**

```
 ...
move BTN-APPLY to lsResourceId
invoke self "getObjectFromId" using lsResourceId
returning lsPushButton
move p2ce-clicked to lsEvent
move dialog-apply to lsLogicalId
invoke lsPushbutton "translateEvent" using lsEvent self
lsLogicalId
*----END-USER.
exit method.
end method "create".
```

- END-USER.

手順 5：コントロールからのデータの設定と取得方法

ダイアログ ボックスのみがコントロールにアクセスできるため、これらのコントロールの中でデータを設定したり、コントロールからデータを呼び出すためのメソッドを作成することが必要です。

1. 値を設定したい各コントロールに対して、そのコントロールにデータを渡すためのメソッドを作成します。
2. データを取得したい各コントロールに対して、そのコントロールのデータを取り込むためのメソッドを作成します。

手順 6：特定のコントロールにフォーカスをセットする方法（オプション）

1. ダイアログ ボックスに"show"メソッドを追加します。

   ダイアログ ボックスのクライアントは、デスクトップに表示するために"show"メッセージを送信します。固有の"show"メッセージを追加することにより、ダイアログのコントロールの状態を変更することができます。

2. "show"メソッドの中で、ダイアログ ボックスが最初に表示されるときにフォーカスを持たせたいコントロールに"setFocus"メッセージを送信します。

   後述の「ダイアログ ボックス &quotshow" メソッドの例」を参照してください。

この時点で固有のダイアログ ボックス サブクラスを書きこみます。アプリケーションで使用するには、必要に応じて、作成、表示、非表示にするために MainWindow クラスにコードを追加する必要があります。

ダイアログ ボックス "show" メソッドの例

このメソッドは、Micro Focus メッセージに送信することにより、osNameField のコントロールにフォーカスを設定するためにダイアログ ボックス"show"メソッドをオーバーライドします。メソッドの最後の文はスーパークラスに"show"メソッドを渡します。それ以外の場合、ダイアログ ボックスのデフォルト"show"コードは実行されず、ダイアログ ボックスはデスクトップ上に表示されません。

- ----------
- **このボックスを表示する前にフィールドをクリアします。**
- ----------

```
method-id. "show".


procedure division.
```

```
invoke osNameField "setFocus"

invoke super "show"

exit method.

end method "show".
```

ステップ 4：ダイアログ ボックスの制御方法

以下の手順は、GUI アプリケーション テンプレートの MainWindow からのダイアログ ボックスの制御に適用されます。

- 手順1： MainWindow クラスのClass-control段落にクラス名を追加します(手順 1：Class-control段落に対するクラスの追加方法参照)。
- 手順2： ダイアログ ボックスを作成し、表示するためのメソッドを追加します(手順 2：ダイアログ ボックスを作成するためのメソッドの追加方法参照)。

  このメソッドはまた、MainWindow クラスのメソッドにダイアログ ボックス イベントをマッピングし、ダイアログ ボックスに初期データを設定します。
- 手順3： ダイアログ イベントを処理するメソッドを追加します。MainWindow クラスには、ダイアログ ボックスをキャンセルするためのデフォルト コードがあります(手順 3：ダイアログ イベントを処理するメソッドの追加方法参照)。

手順 1：Class-control段落に対するクラスの追加方法

1. MainWindow クラスの一番上の近くにあるClass-control段落を捜します（デフォルトでは tpltwin.cbl）。
2. Class-control段落の最後にダイアログ ボックスのクラス名を追加します。USER-DATA と END-USER コメントがガイドとして示されます。

   ```
   class-name is class "filename"
   ```

   パラメータ

class-name	ダイアログ ボックスのクラス名。
filename	このクラスのコードを含むファイル。小文字で入力してください。

このクラスをこのプログラムに認識させるClass-control段落にダイアログ ボックスのクラス名を追加し、メッセージ

を送信できるようにします。

手順 2：ダイアログ ボックスを作成するためのメソッドの追加方法

1. Object 見出しと End-Object 見出しの間の任意の場所で、MainWindow クラスに新規メソッドを追加します。

   後述の「インスタンス メソッドの例」を参照してください。

2. ダイアログ ボックスを作成するためのコードを追加します。ダイアログ ボックスの表示形式は Windows 95 リソース .dll ファイルの中に保持されます。ダイアログ ボックスの動作は、ダイアログ ボックスの Object COBOL クラスにより記述されます。ダイアログ ボックス クラスに"fromResource"メッセージを送信することによって、ダイアログ ボックスのインスタンスを作成します。

```
move dialog-id to lsId
 invoke AboutBoxTemplate "fromResource"
 using parent resource lsId
 returning dialog
```

パラメータ

dialog-Id ダイアログ ボックスを定義するシンボル。Dialog Edlitor を使ってダイアログ ボックスをペイントするときにこれを割り当て、COBOL コピーファイルを通してコードで使用できるようにします。シンボル名の任意のアンダスコア (_) は Object COBOL プログラムで使用するためにハイフン (-) に変更されます。

parent OBJECT REFERENCE
画面上のこのオブジェクトに対する親。通常はメイン ウィンドウ（SELF で表されます）ですが、別のダイアログ ボックスである場合もあります。

lsID PIC X(4) COMP-5.

resource OBJECT REFERENCE
このダイアログ ボックスのリソース モジュール。デフォルトではデータ項目 osResource にあります。

dialog OBJECT REFERENCE.
これは新しく作成されたダイアログ ボックスに対して返されるオブジェクトハンドルを保持します。

インタフェースの状態情報の一部としてオブジェクト ハンドルを維持する必要がない場合、このメソッドのローカル記憶域部で *dialog* を宣言することができます。これは一般的に、モーダル ダイアログ ボックスの場合です。

ダイアログ ボックスの存在がインタフェースの状態情報の一部である場合、オブジェクト記憶域セクションで*dialog*を宣言します。そのようにして、MainWindow クラスのすべてのメソッドで使用することができます。

3. ダイアログ ボックス イベントの対象を登録するためのコードを追加します。

   ダイアログ ボックス イベントは、エンドユーザがプッシュボタンをクリックしたときに論理イベントを生じさせます。他のエンドユーザの動作に対してイベントを定義することもできます。それらに対して動作できるように MainWindow クラスのメソッドにダイアログ ボックス イベントをマップする必要があります。

   各イベントに対して、次のコードが必要です。

```
map event logicalId upon dialog
to method methodname upon self
```

パラメータ

logicalID	ダイアログ ボックス イベントは、コピーファイルで宣言されたレベル-78 データ項目により識別されます（デフォルトでは tpltdial.cpy）。
dialog	OBJECT REFERENCE. ステップ 2 で作成されたダイアログに対するオブジェクト ハンドル。
methodname	メソッドを指定するヌルで終了したリテラルの名前指定。GUI テンプレートで使用されている命名規則にしたがうために、メソッドに名前をつけます。 "onEvent*dialogevent*"

4. ダイアログ ボックスに最初のデータをセットします（オプション）。

   この手順が必要であるかどうかは、ダイアログ ボックスがユーザに対して情報を表示するかどうかによって決まります。情報をセットするためにダイアログ ボックスに対して送信するメッセージは、ダイアログ ボックス クラスで定義されているメソッドによって決まります。

5. ダイアログ ボックスを表示します。

```
invoke dialog "create"
```

```
invoke dialog "show"
```

パラメータ

dialog OBJECT REFERENCE.
ダイアログ ボックスのオブジェクト ハンドル（ステップ 2）。

これで、ダイアログ ボックスの作成と表示に必要なすべてのコードが揃います。

後述の「ダイアログ ボックスの作成と表示の例」を参照してください。

インスタンス メソッドの例

クラスの中のインスタンス メソッドの位置を示します。

```
class-id. Example inherits
          from Base.
* 環境部と Class-control 節
 ...
* 共有データ

...
* クラス オブジェクト コード
 ...
* インスタンス オブジェクト コード

object.
```

- **インスタンス データ**

```
...
```

```
method-id. "setItem1".
* メソッド コードとデータ
...
end method "setItem1".
```

- **さらにメソッドが続きます**

```
...
end object.
end class Example.
```

関連項目

- 12.5.4 インスタンス メソッド

ダイアログ ボックスの作成と表示例

次のコードは、AccountAdder クラスのダイアログ ボックスを作成します。このダイアログ ボックスのリソース ID は、DLG-OPEN-ACCOUNT により表されます。これは、このクラスのリソース コピーファイルで宣言されたレベル 78 データ項目です。

"onDialogCancel"メソッドは、dialog-cancel イベントにマップされ、MainWindow クラスの一部として指定されるダイアログ ボックスをキャンセルするためのデフォルト メソッドです。 "onDialogAccountOk"メソッドは、dialog-ok メソッドにマップされ、アプリケーションに特有です。

```
method-id. "onEventNewAcc".
local-storage section.
01 lsDialog              object reference.
01 lsId                  pic x(4) comp-5.
01 lsEvent               pic x(4) comp-5.

procedure division.
```

```
move DLG-OPEN-ACCOUNT to lsId
invoke AccountAdder "fromResource" using self osResource lsId
returning lsDialog
move dialog-ok to lsEvent
invoke lsDialog "setEventTo" using lsEvent self
z"onDialogAccountOk"
move dialog-cancel to lsEvent
invoke lsDialog "setEventTo" using lsEvent self
z"onDialogCancel"
invoke lsDialog "create"
invoke lsDialog "show"
exit method.
end method "onEventNewAcc".
```

手順 3：ダイアログ イベントを処理するメソッドの追加方法

1. . 操作する各ダイアログ イベントのメソッドを追加します。

   イベントで起動したメソッドは、必ずイベント オブジェクトをパラメータとして受信します。イベント オブジェクトに"getObject"メッセージを送信し、イベントを起こすダイアログ ボックスのオブジェクト ハンドルを取得します。

   ```
   invoke lnkEvent "getObject"
                   returning lsDialog
   ```

   パラメータ

   lsDialog OBJECT REFERENCE.
   これを、このメソッドのローカル記憶域セクションで宣言します。

lnkEvent OBJECT REFERENCE
このメソッドのLinkage Sectionで宣言します。これは、USING パラメータとして渡されます。

2. メッセージをダイアログ オブジェクトに送信して、その情報を検索します。送信するメッセージは、情報を検索するダイアログ ボックスで定義されたメソッドにより異なります。

3. ダイアログ ボックスを破棄する場合は、これに"hide"および"finalize"メッセージを送信します。

```
invoke lsDialog "hide"

invoke lsDialog "finalize"

             returning lsDialog
```

パラメータ

lsDialog OBJECT REFERENCE
手順 1 で検索したオブジェクト ハンドルです。

後述の「ダイアログ ボックスからイベントを処理するメソッドの例」を参照してください。

これにより、ダイアログ ボックスを表示し、そこからイベントを処理し、これを破棄するために必要な、最小限の手順を完了できます。

ダイアログ ボックスからイベントを処理するメソッドの例

下のメソッドは、情報を問い合わせるダイアログ ボックスを表示し、データはシグナルに情報として送信するためにSymbolTable に格納されます。

ダイアログ ボックスのオブジェクト ハンドルは、メソッドに渡されるイベント オブジェクトから検索されます。すべての情報が回復されると、ダイアログ ボックスは"finalize"メッセージをそれに送信することにより破棄されます。

```
method-id. "onDialogAccountOk".

local-storage section.

01 lsTable              object reference.

01 lsDialog             object reference.

01 lsString             object reference.
```

```
linkage section.
01 lnkEvent              object reference.
procedure division using lnkEvent.
```

- ---SymbolTableを作成します

```
   invoke SymbolTable "new" returning lsTable
```

- ---ダイアログ ボックスのハンドルを呼び出します

```
   invoke lnkEvent "getObject" returning lsDialog
```

- ---ダイアログ ボックスからデータを取り出します

```
   invoke lsDialog "getName" returning lsString
```

- ---SymbolTableにデータを入れます

```
   put value lsString into table lsTable
at symbol accountName-key
invoke lsDialog "getBalance" returning lsString
put value lsString into table lsTable
at symbol accountBalance-key
invoke lsDialog "getType" returning lsString
put value lsString into table lsTable
```

```
at symbol accountType-key
```

- ---ダイアログ ボックスを非表示にし、最終処理します

```
   invoke lsDialog "hide"
invoke lsDialog "finalize" returning lsDialog
```

- ---シグナルを送信します

```
   invoke self "signal" using open-account lsTable
```

- ---テーブルと関連付けられたデータを最終処理します

```
   invoke lsTable "deepFinalize" returning lsTable
exit method.
end method "onDialogAccountOk".
```

ステップ 5：イベントで起動されるメソッドの作成方法

イベントで起動されるメソッドを作成するには、次の手順を行います。

1. 手順1： イベントをメソッドにマップします(手順 1：イベントのメソッドへのマップ方法参照)。
   イベントが起こるたびに、メソッドが起動されます。

- 手順2： メソッドを作成します(手順 2：イベントを受け取るメソッドの作成方法参照)。
- 手順3： イベントを問い合わせます(手順 3：イベントの問い合わせ方法参照)。

必ずこれを行う必要はありません。予期しているイベントが起こっていることを、たびたび認識するようにすれば充分です。ただし、さらに情報を得る必要があることもあります。たとえば、異なるオブジェクトには、エイベン

トを問い合わせてイベントを引き起こすオブジェクトを検出する場合に、このメソッドにマップされたイベントをすべて持つものもあります。

MainWindowクラス (tpltwin.cbl) の "onDialogCancel" メソッドは、このように動作します。

手順 1：イベントのメソッドへのマップ方法

1. イベント通知先のオブジェクトを識別します。

   ダイアログ ボックスからMainWindowメソッドにイベントをマッピングするには、非常に時間がかかります。各ダイアログ ボックスで、コントロールからのイベントに対応する論理イベントを定義します。たとえば、dialog-OK および dialog-cancel は、ユーザの押す 「はい」 および 「キャンセル」 ボタンに対応します。ダイアログ ボックス論理イベントは、tpltdial.cpy でレベル 78 項目として定義されます。

   他の場合は、クライアント領域から物理イベントをマッピングしていることもあります。物理イベントは、すべてコピーファイル p2cevent.cpy でレベル 78 項目として定義されます。

2. 各新規メニュー項目のマップ文を追加して、メニュー項目の識別子をMainWindowクラスのメソッドにリンクさせます。これは、エンド ユーザがメニュー項目を選択する際に必ず実行されるものです。

   ```
   map event eventId upon object


       to method method upon self
   ```

   パラメータ

eventId	イベントの数値識別子です。イベントは、通常コピーファイルでレベル 78 項目として定義されます。
object	OBJECT REFERENCE それ自体のイベントを受け取るオブジェクトです。
method	ヌルで終わる、メソッドのリテラル命名です。GUI テンプレートで使う命名規則に従って、次のようにメソッドに名前を付けます。 "onEvent*eventname*" または "onDialog*eventname*"

手順 2：イベントを受け取るメソッドの作成方法

MainWindowクラスで、Object見出しとEnd Object 見出しの間のいずれかの場所に、新規のメソッド (tpltwin.cbl) を追加します。

```
 method-id. "method".


linkage section.
 01 lnkEvent          object reference.
 procedure division using lnkEvent.
...   *> イベントを処理するコード
    exit method.
 end method "method".
```

パラメータ

method	直前の手順のシグナルにマップしたメソッドの名前

手順 3：イベントの問い合わせ方法

以下のどのステップを実行するかは、イベントに関して必要な情報量により異なります。

1. イベントを引き起こしたオブジェクトの検索方法：

```
 invoke lnkEvent "getObject"
           returning object
```

パラメータ

object	OBJECT REFERENCE. イベントを引き起こしたオブジェクトの操作

2. イベントの数値 ID を検索するには

```
invoke lnkEvent "getId"
          returning Id
```

パラメータ

Id PIC X(4) COMP-5. .
イベントの数値 ID

3. 情報のほかのタイプは、イベント タイプに依存します。イベントのほかのタイプ、およびそれらの問い合わせ方法を見るには、次の関連項目を参照してください。

関連項目

- コマンド イベント クラス
- キーボード イベント クラス
- マウス イベント クラス
- 移動イベント クラス/LI
- スクロール イベント クラス
- 選択イベント クラス
- サイズ イベント クラス

ステップ 6：情報のクライアント領域への設定方法

クライアント領域に情報を表示するオプションには、主に以下の 2 つがあります。

- オプション 1： リスト ボックスや複数行エントリ フィールドのようなコントロールを使います。
- オプション 2： サブペインへのペイント

オプション 1：コントロールを使った表示方法

- 手順1： コントロールのクラスを、MainWindowのClass-control段落に追加します(手順 1：クラスのClass-control段落への追加方法参照)。
- 手順2： データを表示するコントロールのタイプを指定するには、MainWindowクラスで「クライアント ペインの定義」メソッドを上書きします(手順 2：コントロール タイプの指定方法参照)。

  デフォルトで、メイン ウィンドウのクライアント領域はサブペイン オブジェクトですが、これを変更して、リスト ボックス、複数行エントリ フィールド、またはその他の適切なコントールにすることができます。
- 手順3： コントロールに必要なイベント操作を設定します(手順 3：イベント操作の設定方法参照)。

  たとえば、リスト ボックスを使おうとしている場合は、エントリでダブル クリックするよう選択できます。
- 手順4： メソッドを作成して、データをコントロールに設定します(手順 4：データのコントロールへの設

定方法参照)。

手順 1：クラスのClass-control段落への追加方法

1. Class-control段落をMainWindowクラス（デフォルトでtpltwin.cbl）の一番上近くに配置します。
2. Class-control段落に、コントロールのエントリがすでにあるかどうか、注意して見ます。

   なかった場合は、コントロールのクラス名をClass-control段落の最後に追加します。この段落は、ガイドとして提供されたUSER-DATA とEND-USER のコメントの間にあります。

   ```
   class-name is class "filename"
   ```

   パラメータ

   class-name コントロールのクラス名です。

   filename このクラスのコードを含むファイルです。小文字で入力します。

ダイアログ ボックスのクラス名をClass-control段落に追加すると、このクラスがこのプログラムに認識され、これにメッセージを送信できます。クラス ライブラリのリファレンスから、すべてのグラフィック クラスのクラス名とファイル名を検索できます。

手順 2：コントロール タイプの指定方法

1. MainWindowクラス（tpltwin.cbl）で、"defineClientPane"メソッドを検索します。
2. 以下のようにして、クライアント領域オブジェクトを作成する文を編集します。

   ```
   invoke control "new" using lnkParent
                  returning lnkPane
   ```

   パラメータ

   control 使用するコントロールのクラス名です。

手順 3：イベント操作の設定方法

以下の文を、作用させる各イベントに追加します。

```
invoke lnkPane "setEventTo"
               using lsId self
```

z"*method*"

パラメータ

method 起動するメソッドの名前です。

クライアント領域の例

下のコードはリストボックスとして GUI アプリケーション テンプレートのクライアント領域を定義し、"onEventListDoubleClick"メソッドに対してエントリ上でエンドユーザがダブルクリックすると発生するイベントをマップします。

```
method-id. "defineApplicationSpace".
linkage section.
01 lnkParent             object reference.
01 lnkSpace              object reference.
procedure division using lnkParent returning lnkSpace.
```

- ---USER-CODE. **代わりのクライアント領域のクラスを指定します**

- **下記の文のサブペイン**

```
invoke Listbox "new" using lnkParent returning lnkSpace
```

- ---USER-CODE. **下のクライアント領域のイベントハンドリングを追加します。**

```
invoke lnkSpace "setEventTo"
using lsId self z"onEventListDoubleClick"
```

- ---END-USER.

```
    exit method.
end method "defineApplicationSpace".
```

手順 4：データのコントロールへの設定方法

1. MainWindowクラス（tpltwin.cbl）で、新規メソッドを作成します。

   表示するデータの発信元に従って、このメソッドは、MainApplicationクラス（tpltapp.cbl）によって、またはエンド ユーザ イベントの結果として起動されます。適切なイベントまたはシグナルにメソッドが確実にマップされるようにして、データ表示が必要なときに更新されるようにする必要があります。

2. メソッドで、以下のようにしてコントロールのオブジェクト ハンドルを検索します。

   ```
   invoke self "getClientPane"
               returning lsPane
   ```

   パラメータ

lsPane	OBJECT REFERENCE. ローカル記憶データ項目を宣言して、ハンドルをクライアント領域コントロールに保存します。

3. メッセージを送信して、データをコントロールに渡します。

クライアント領域 コントロールのデータの設定例

下のメソッドは表示するデータを含む SymbolTable をそれに渡す MainApplication オブジェクトのシグナルの結果として呼び出されます。クライアント領域はリストボックスとして定義され、渡されるデータは CharacterArrays の OrderedCollection です。

```
method-id. "onSignalShowTypeList".
local-storage section.
01 lsCollection          object reference.
01 lsSpace               object reference.
```

```
linkage section.
01 lnkDataBlock           object reference.

procedure division using lnkDataBlock.
```

- ---SymbolTableからデータを取得します。

```
   fetch value lsCollection from table lnkDataBlock
at symbol displayCollection-key
```

- ---クライアント領域 オブジェクトのオブジェクト ハンドルを検索します。

```
   invoke self "getApplicationSpace" returning lsSpace
```

- ---クライアント領域のコレクションを表示します。

```
   invoke lsSpace "setContents" using lsCollection
exit method.
end method "onSignalShowTypeList".
```

オプション 2：サブペインへのペイント

- 手順1： Subpaneからサブクラス化して、ユーザ独自のサブペイン クラスを作成します(手順 1： Subpane のサブクラスの作成方法参照)。
- 手順2： サブペイン クラスをMainWindow のClass-control段落に追加します（手順 2：サブクラスの MainWindowのClass-control段落への追加方法参照）。

- 手順3： MainWindowクラスにある"defineClientPane"メソッドに、サブペイン クラスを指定します(手順 3：クライアント ペインとしてのサブペインの使用方法参照)。
- 手順4： "paint" メソッドをサブクラスに追加します(手順 4"paint" メソッドの上書き方法参照)。

手順 1： Subpaneのサブクラスの作成方法

1. 新規プログラムを作成し、以下のテキストを入力します。

```
class-id. class-name
          inherits from Subpane.


 class-control.
    TextDisplay is class "textdisp"
    SubPane is class "subpane"
    .
 object.
 object-storage section.
 end object.
 end class class-name.
```

パラメータ

class-name　　サブクラスの名前です。

filename　　このクラスのコードを含むファイルです。小文字で入力します。

2. 新規プログラムをプロジェクトに追加します。

手順 2：サブクラスのMainWindowのClass-control段落への追加方法

1. Class-control段落をMainWindowクラス（デフォルトでtpltwin.cbl）の一番上近くに配置します。
2. ガイドとして提供されている ユーザ データとエンド ユーザのコメントの間にあるClass-control段落の最後に、サブペイン クラスのクラス名を追加します。

```
class-name is class "filename"
```

パラメータ

class-name　　サブペイン クラスのクラス名です。

filename　　このクラスのコードを含むファイルです。小文字で入力します。

手順 3：クライアント ペインとしてのサブペインの使用方法

1. MainWindowクラス（tpltwin.cbl）で、"defineClientPane"メソッドを検索します。
2. クライアント領域オブジェクトを作成する文を編集します。

```
invoke classname "new" using lnkParent
                returning lnkSpace
```

パラメータ

classname　　サブペイン クラスのクラス名です。

手順 4： "paint"メソッドの上書き方法

1. SubpaneのサブクラスでObject見出しとEnd Object 見出し間に、以下の文を入力します。

```
method-id. "paint".
linkage section.
01 lnkSurface     object reference.
procedure division using lnkSurface.
```

- Code to paint onto the surface.

```
    exit method.
  end method. "paint"
```

このメソッドで、サブペインを再ペイントするウィンドウをいつでも実行できます。テキスト、ビットマップ、およびシェイプの任意の組み合わせを、パラメータとして渡されたDrawingSurfaceオブジェクトに配置できます。 描画面にペイントするコードに関する詳細については、関連項目を参照してください。

ウィンドウのクライアント領域へのペイント

ペイントを使用すると柔軟な表示が可能となり、任意のウィンドウやガジェット上に次の表示ができます。

- ウィンドウ システムで利用可能な任意のフォントのテキスト
- ビットマップ
- 塗りつぶし図形 (単色またはビットマップによる塗りつぶし)
- ライン

このような表示を行うには、ペイントするオブジェクトの型のサブクラス (AbstractWindow の任意のサブクラスで構いませんが、一般的にはサブペイン) を作成して、"paint"メソッドを無効にします。"paint"メソッドはウィンドウ システムがオブジェクトのすべてまたは一部を描く必要があるごとに起動され、DrawingSurface がパラメータとして渡されます。

DrawingSurface への描込みを行うには、次のタイプのオブジェクトを 1 つ以上作成する必要があります。

- フォント (テキスト用)
- ペン (ラインおよび曲線用)
- ブラシ (塗りつぶし図形用)

作成された各フォント、ペン、またはブラシは、限りのある、Windows 95 上の Windows GDI リソースを使用します (詳細については、Windows SDK のドキュメントを参照してください)。 このため、必要なときにペンやブラシを作成し、使用し終わったらそのペンやブラシをすぐに削除する ("finalize"メッセージを送信する) ことをお勧めします。 一般的には、"paint"メソッドの起動時にペンやブラシを作成し、終了時に削除します。

この方法ではアプリケーションが遅くなる場合は、ペンやブラシをObject-Storageにキャッシュすることができます。ただしこの場合も、ペンやブラシが不要になった場合には削除するよう注意が必要です。 たとえば、青のペンを使用していたときに、エンドユーザがそのペンで描かれていたオブジェクトを赤で描くことにした場合は、青のペンを削除してから赤のペンを作成します。 ペンやブラシを、古いものを削除しないで作成し続けると、Windows 95 の GDI 資源が不足します。

フォントの作成には、ペンの作成よりも時間がかかります。 また、一般的なアプリケーションでは、フォントをたくさん使用することはありません。 このため、各「ペイント」ごとにフォントを作成し削除するよりは、Object-Storage にフォントをキャッシュすることをお勧めします。

場合によっては、Windows にウィンドウを再ペイントさせる必要があります。 "paint"メソッドが起動されるごとに、

ウィンドウ上の描込みを行った領域だけが再ペイントされます。 ウィンドウ上に、以前に起動された"paint"メソッドによる他のオブジェクトが存在しても、それらのオブジェクトは更新されません。Windows にウィンドウを完全に再ペイントさせるには、ウィンドウに"invalidate"メッセージを送信します。 "invalidate"メッセージは、"paint"メソッド内でなければ、どこからでも送信できます。 "paint"メソッド内でこのメッセージを送信すると、ウィンドウがちらつきます。

関連項目

- Fontクラス
- Brushクラス
- DrawingSurfaceクラス

テキストのペイント例

次のコード例では、ウィンドウの左上に 10 ポイントの Arial テキストで "Hello world" を挿入するよう上書きされる "paint" メソッドを示します。このメソッドは、一度作成されると object-storage にフォント オブジェクトを一時保存します。

```
   method-id. "paint".
   local-storage section.
   01 lsFlags               pic x(4) comp-5.
   01 lsSize                pic x(4) comp-5.
   01 lsY                   pic x(4) comp-5.
   01 lsX                   pic x(4) comp-5.
   01 lsString              object reference.
   01 lsBool                pic x comp-5.
    88 isTrue               value 1.
    88 isFalse              value 0.


linkage section.
01 lnkDrawingSurface       object reference.
procedure division using lnkDrawingSurface.
```

- ---フォント オブジェクト (object-storage で宣言されると仮定される osFont) を作成します。

```
if osFont = null
   invoke CharacterArray "withValue" using z"Arial"
                          returning lsString
   move 10 to lsSize
   move 0 to lsFlags
   invoke Font "new" using lsString lsSize lsFlags
               returning osFont
   invoke lsString "finalize" returning lsString
end-if
```

- ---フォントを DrawingSurface に設定します。

```
invoke lnkDrawingSurface "setFont" using osFont
invoke CharacterArray "withValue" using z"Hello World"
                         returning lsString
move 2 to lsX
move 2 to lsY
```

- ---テキストを表示します。

```
invoke lnkDrawingSurface "drawTextAt" using
```

```
                                    lsString lsX lsY
       invoke lsString "finalize" returning lsString
       exit method
   end method "paint".
```

ラインのペイント例

次のコード例では、ウィンドウのサイズや面積に比例して楕円を描くよう上書きされる "paint" メソッドを示します。

```
   method-id. "paint".
   local-storage section.
   01 lsFlags               pic x(4) comp-5.
   01 lsSize                pic x(4) comp-5.
   01 lsY                   pic x(4) comp-5.
   01 lsX                   pic x(4) comp-5.
   01 lsX1                  pic x(4) comp-5.
   01 lsY1                  pic x(4) comp-5.
   01 lsPen                 object reference.
   01 lsOldPen              object reference.
   01 lsColor               object reference.
   01 lsBool                pic x comp-5.
    88 isTrue               value 1.
    88 isFalse              value 0.

   linkage section.
   01 lnkDrawingSurface      object reference.
```

```
procedure division using lnkDrawingSurface.
```

- ---ペンの色を作成します。

```
 invoke Color "blue" returning lsColor
```

- ---ペンを作成します。

```
 invoke Pen "new" returning lsPen
 invoke osPen "setColor" using lsColor
 invoke lsColor "finalize" returning lsColor
```

- ---DrawingSurface でペンを設定します。

```
 invoke lnkDrawingSurface "setPen" using lsPen
                          returning lsOldPen
 invoke self "getClientWidthHeight" using lsX lsY
 divide lsX by 2 giving lsX1
 divide lsY by 2 giving lsY1
 divide lsX by 3 giving lsX
 divide lsY by 3 giving lsY
```

- ---楕円を描きます。

```
 invoke lnkDrawingSurface "ellipse" using lsX1 lsY1 lsX lsY
```

- ---**直前のペンを復元します。**

```
invoke lnkDrawingSurface "setPen" using lsOldPen
                         returning lsPen
```

- ---**新しいペンを破棄します。**

```
invoke lsPen "finalize" returning lsPen
exit method
end method "paint".
```

ステップ 7：シグナルの処理方法

- 手順1： シグナルをメソッドにマップします(手順 1：シグナルのマップ方法参照)。
- 手順2： シグナルを受信するメソッドを作成します(手順 2：シグナルを受け取るメソッドの作成方法参照)。
- 手順3： SymbolTableの内容をアンパックします(手順 3：SymbolTable内容のアンパック方法参照)。

手順 1：シグナルのマップ方法

1. "defineInSignals" メソッドを、MainWindowクラス（tpltwin.cbl）またはMainApplicationクラス（tpltapp.cbl）のいずれかに配置します。
2. 各シグナルに以下の文を追加します。

```
map signal signal-name from
   socket socket-name to method method
```

パラメータ

signal-name ヌルで終わる定数で、受信するシグナルに名前を付けます。メイン アプリケーション、および GUI アプリケーション テンプレートのメイン ウィンドウ コンポーネントは、コピーファイル tpltapp.cpy および tpltwin.cpy で個別にレベル 78 として定義されます。

socket-name ヌルで終わる定数で、これらのシグナルを受信するために定義されるソケットに名前を付けます。ソケットは、MainApplicationクラスまたは GUI アプリケーション テンプレートのメイン ウィンドウ コンポーネントからのシグナルに付けたソケット名は、コピーファイル tpltapp.cpy および tpltwin.cpy で個別にレベル 78 として定義されます。
メイン アプリケーション コンポーネントからのシグナルの場合は、メイン アプリケーション ソケットをソケット名として使います。メイン ウィンドウ コンポーネントからのシグナルの場合は、ウィンドウ ソケットをソケット名として使います。

method ヌルで終わる定数で、シグナルの受信の際に起動されるメソッドに名前を付けます。GUI テンプレートで使う命名規則に従って、次のようにメソッドに名前を付けます。

"onSignal*name*"

手順 2：シグナルを受け取るメソッドの作成方法

シグナルを受信中のクラスで、Object 見出しとEnd Object 見出し間の任意の場所に、新規メソッドを追加します。

```
method-id. "methodname".


linkage section.
01 lnkData         object reference.
procedure division using lnkData.
... *> Code to unpack and process data
   exit method.
end method "methodname".
```

パラメータ

methodname　　Tシグナルにマップされたメソッドの名前です。

手順 3：SymbolTable内容のアンパック方法

1. SymbolTableの各データの場合

```
fetch value lsObject from table lnkData
                at symbol symbolname
```

パラメータ

lsObject　　OBJECT REFERENCE.
データとして渡されたCharacterArrayまたはOrderedCollectionは、データ項目に入れられます。

lnkData　　OBJECT REFERENCE.
シグナルに従って渡されたSymbolTableです。

symbolname　　検索するデータのシンボルです。

2. 予期したデータが処理前にSymbolTableにあったかどうか、試験することができます。lsObject にヌル ハンドルが含まれている場合は、データは存在していません。

```
if lsObject <> null
```

- **処理データ**

```
end-if
```

この手順はオプションです。

例については、前述のステップ6、オプション1の「手順 4：データのコントロールへの設定方法」にある「クライアント領域コントロールのデータの設定例」を参照してください。

25.3.6 ビジネス ロジックの作成

GUI アプリケーション テンプレートのビジネス ロジックは、ユーザ インタフェースから受信したシグナルにより動作します。MainApplication クラス (tpltapp.cbl) は、インタフェースとビジネス ロジックの間のリンクであり、受信したシグナルを解読し、ビジネス ロジックで他のほかのプログラムが要求を処理するように呼び出します。

MainApplication クラスはまた、シグナルを通してインタフェースに情報を返すことにも責任をもっています。一般的なサイクルは、次のとおりです。

- ステップ1： シグナルを受信し、要求を処理します。
  後述のステップ 1：シグナルの処理方法を参照してください。
- ステップ2： 処理された要求を受諾するシグナルを送信します。
  後述のステップ 2：シグナルの送信方法を参照してください。

受諾シグナルは、表示するインタフェースのデータも搬送します。

オブジェクト指向または手続きコードを使用したアプリケーションのビジネス ロジックを作成することができます。ビジネス ロジックの残りの部分がオブジェクトである場合、MainApplication はインタフェースから受信した情報を渡しながら、それに対してメッセージを送信します。

ビジネス ロジックが一つまたは複数の手続き COBOL プログラムにある場合、MainApplication が呼び出すことができるエントリポイントを指定する必要があります。ビジネス ロジックの各ルーチンは一つの要求を処理することが要求されるため、EXIT PROGRAM 文をもつ MainApplication クラスにコントロールを返します。

このような GUI アプリケーションのビジネス ロジックを作成することは、CICS などのトランザクション処理システムの下でビジネス ロジックを作成することと似ています。その際、各トランザクションは機能する別のコードによりハンドルされてから、終了します。トランザクションは処理機能を利用できないような短い期間で行われることが必要なCICS システムとまったく同様、GUI アプリケーションのトランザクションはエンドユーザのインタフェースをロックアップしないように短い期間で行われることが要求されます。

ステップ 1：シグナルの処理方法

25.3.5「ユーザ インタフェースの作成」の「ステップ 7：シグナルの処理方法」 を参照してください。

ステップ 2：シグナルの送信方法

- 手順1： SymbolTableを作成して、シグナルを使って送信するデータを保存します(手順 1：SymbolTableの作成方法参照)。
- 手順2： SymbolTableにデータを書き込みます(手順 2：SymbolTableへのデータの書き込み方法参照)。

- 手順3： シグナルを送信します(手順 3：シグナルの送信方法参照)。
- 手順4： SymbolTableおよびそのデータを最終化します(手順 4:SymbolTableの最終化方法参照)。

手順 1：SymbolTableの作成方法

1. OBJECT REFERENCE型のデータ項目を宣言して、オブジェクト ハンドルをSymbolTableに保存します。
2. 一般に、メソッドの開始時にSymbolTableを作成し、終了時に破棄するので、メソッド ローカル記憶域セクションでこれを宣言します。
3. 以下のようにして、"new" メッセージをSymbolTable クラスに送信します。

```
invoke SymbolTable "new"
      returning table
```

パラメータ

table OBJECT REFERENCE
新規のSymbolTableへのオブジェクト ハンドルです。

手順 2：SymbolTableへのデータの書き込み方法

1. SymbolTableに書き込むデータの各項目は、CharacterArrayまたはOrderedCollectionのいずれかになります。データがまだこのフォーマットにない場合は、このフォーマットにデータを書き込む必要があります。
2. オブジェクト ハンドルをSymbolTableのデータに格納します。

```
put value object into table table
        at symbol symbol
```

パラメータ

object OBJECT REFERENCE
格納するCharacterArrayまたはOrderedCollectionです。

table OBJECT REFERENCE
この手順の直前のトピックで作成した、SymbolTableへのオブジェクト ハンドルです。

symbol ヌルで終わる定数で、ヌルも含めて 32 文字以内の長さです。
データ格納先のシンボルです。アプリケーション テンプレートにより、コピーファイル（デフォルトでtpltkeys.cpy）でレベル 78 としてシンボルが定義されます。

CharacterArrayの作成方法

CharacterArray を作成する多くの方法があります。 2 つの最も便利な方法は次のとおりです。

- "withValue" メソッドを使用する

```
invoke CharacterArray "withValue"
 using zstring
 returning aCharArray
```

パラメータ

zString 任意のヌルで終了する文字列

aCharArray OBJECT REFERENCE.

- "withLengthValue"メソッドを使用する

```
invoke CharacterArray "withLengthValue"
 using aLength aLiteral
 returning aCharArray
```

パラメータ

aLiteral 任意の COBOL リテラル、またはそれを含むデータ項目。

aLength PIC X(4) COMP-5.
*aLiteral*の文字の長さ。

aCharArray OBJECT REFERENCE

オブジェクトを格納するCharacterArray の作成方法

OrderCollection クラスに"ofReferences" メッセージを送信します。

```
invoke OrderedCollection "ofReferences"
            using aLength

 returning aCollection
```

パラメータ

aLength PIC X(4) COMP-5.
OrderedCollection の最初のサイズを含みます。 制限を越えた場合、コレクションが自動的に大きくなります。

aCollection OBJECT REFERENCE
新しく作成されたOrderedCollectionsに対するオブジェクト ハンドルを含みます。

手順 3：シグナルの送信方法

次のコードを使います。

```
signal signalname table
```

パラメータ

signalname ヌルで終わる定数で、送信するシグナルに名前を付けます。
これらは、コピーファイル (デフォルトでtpltwin.cpy) でレベル 78 としてMainWindowクラスを使って定義されます。

table OBJECT REFERENCE
SymbolTableへのオブジェクト ハンドルです。

手順 4:SymbolTableの最終化方法

SymbolTableだけではなくそのデータ項目もすべて最終化して、メモリリークを防止する必要があります。

```
invoke table "deepFinalize"
            returning table
```

パラメータ

table	OBJECT REFERENCE SymbolTableへのオブジェクト ハンドルです。

ダイアログ ボックスからイベントを操作するメソッドの例

下のメソッドは、情報を問い合わせるダイアログ ボックスを表示し、データはシグナルに情報として送信するためにSymbolTable に格納されます。

ダイアログ ボックスのオブジェクト ハンドルは、メソッドに渡されるイベント オブジェクトから検索されます。すべての情報が回復されると、ダイアログ ボックスは"finalize"メッセージをそれに送信することにより破棄されます。

```
method-id. "onDialogAccountOk".
local-storage section.
01 lsTable               object reference.
01 lsDialog              object reference.
01 lsString              object reference.

linkage section.
01 lnkEvent              object reference.
procedure division using lnkEvent.
```

- ---SymbolTableを作成します

```
invoke SymbolTable "new" returning lsTable
```

- ---ダイアログ ボックスのハンドルを呼び出します

```
   invoke lnkEvent "getObject" returning lsDialog
```

- ---ダイアログ ボックスからデータを取り出します

```
   invoke lsDialog "getName" returning lsString
```

- ---SymbolTableにデータを入れます

```
   put value lsString into table lsTable
at symbol accountName-key
invoke lsDialog "getBalance" returning lsString
put value lsString into table lsTable
at symbol accountBalance-key
invoke lsDialog "getType" returning lsString
put value lsString into table lsTable
at symbol accountType-key
```

- ---ダイアログ ボックスを非表示にし、最終処理します

```
   invoke lsDialog "hide"
invoke lsDialog "finalize" returning lsDialog
```

- ---シグナルを送信します

```
    invoke self "signal" using open-account lsTable
```

- ---テーブルと関連付けられたデータを最終処理します

```
    invoke lsTable "deepFinalize" returning lsTable
exit method.
end method "onDialogAccountOk".
```