helion.pl
KSIĘGARNIA INTERNETOWA WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

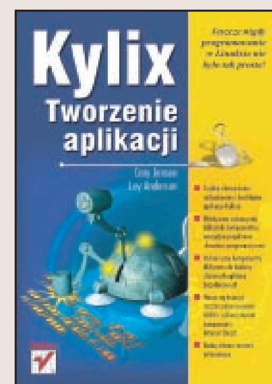
KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

Kylix. Tworzenie aplikacji



Autorzy: Cary Jensen, Loy Anderson
Tłumaczenie: Daniel Kaczmarek (roz. Wprowadzenie, 1- 5, 15 - 17, 19 - 21), Piotr Pilch (roz. 6 - 12), Anna Czerwińska (roz. 13,14), Piotr Tęczyński (roz. 18)
ISBN: 83-7197-651-8
Tytuł oryginału: Building Kylix Applications
Format: B5, stron: 714
Przykłady na ftp: 433 kB

Kylix to jedno z najbardziej oczekiwanych narzędzi programistycznych ostatnich lat. Programistom piszącym programy dla systemu Linux brakowało narzędzia RAD, pozwalającego na szybkie, wizualne projektowanie i tworzenie gotowych aplikacje. Stąd pojawienie się „Delphi dla Linuksa", jak można nazwać Kylix, przyjęte zostało z dużym zainteresowaniem. Dzięki Kyliksowi można nie tylko tworzyć potężne serwery dla Linuksa, ale też aplikacje posiadające graficzny interfejs użytkownika.

Książka „Kylix. Tworzenie aplikacji" jest przewodnikiem dla programisty, w którym znany ekspert w dziedzinie Kyliksa, Cary Jensen, prezentuje wszystko, co niezbędne, by w pełni zapanować nad tym najnowszym narzędziem typu RAD — począwszy od podstawowych czynności programistycznych, a skończywszy na wdrażaniu aplikacji przeznaczonych dla sieci WWW. Nauczysz się więc:

- Posługiwać się zintegrowanym środowiskiem programistycznym Kyliksa
- Konfigurować i rozszerzać możliwości edytora kodu źródłowego
- Usuwać błędy z programów
- Wykorzystywać biblioteki CLX
- Tworzyć aplikacje bazodanowe w Kyliksie
- Pisać aplikacje internetowe działające po stronie serwera
- Tworzyć własne serwery internetowe

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

O Autorach **13**

Wprowadzenie **15**

Część I Tworzenie aplikacji w Kyliksie **19**

Rozdział 1. Wprowadzenie do Kyliksa **21**
Przegląd Kyliksa 21
Tworzenie aplikacji przy użyciu komponentów 22
Kylix generuje kod źródłowy 25
Kylix jako środowisko programowania sterowane zdarzeniami 27
Programowanie zorientowane obiektowo 28
Open Tools API w Kyliksie 28
Czym jest Rapid Application Development? 29
Kylix dla programistów Delphi 29
Kylix i Linux 31
Przegląd CLX 32
Pułapki i sztuczki 34

Rozdział 2. Tworzenie aplikacji **39**
Tworzenie najprostszej aplikacji 39
Tworzenie nowego projektu 40
Umieszczanie i konfigurowanie komponentów 43
Procedury obsługi zdarzeń 45
Dodawanie menu 48
Wykorzystanie dodatkowych form i okien dialogowych 51
Uruchamianie projektu z wiersza poleceń 62
Pliki projektu Kyliksa 64
Plik źródłowy projektu 64
Moduł 65
Plik formy 66
Moduł skompilowany 66
Plik wykonywalny 66
Pliki zmodyfikowane 67
Plik ustawień projektu 67
Plik ustawień kompilatora projektu 67

Rozdział 3. Architektura RAD Kyliksa **69**
Wizualne dziedziczenie form 69
Zmiana właściwości obiektów dziedziczonych 73
Pokrywanie procedur obsługi zdarzeń w obiektach potomnych 73
Dziedziczenie form z projektu bieżącego 76
Definiowanie współdzielonego repozytorium obiektów 77

Projektowanie list akcji ... 77
Praca z ramkami ... 81
Tworzenie ramki ... 81
Wykorzystanie ramek ... 83
Pokrywanie właściwości komponentów osadzonych na kontenerze ... 84
Procedury obsługi zdarzeń obiektów w ramce ... 86
Pokrywanie procedur obsługi zdarzeń obiektów w ramce ... 88
Ramki i zasoby ... 89
Uproszczenie pracy z ramkami ... 91
Przekształcanie ramek w komponenty ... 93
Praca z modułami danych ... 94
Współużytkowanie komponentów z modułami danych ... 95
Ograniczenia modułów danych ... 96

Rozdział 4. Wykorzystanie i konfiguracja edytora ... 99
Zbiory odwzorowań klawiszy edytora ... 100
Uzyskiwanie pomocy na temat odwzorowań klawiszy ... 101
Wybrane kombinacje klawiszy edytora ... 102
Nagrywanie makr klawiszy ... 102
Wstawianie i usuwanie wcięć w blokach tekstu ... 103
Wykorzystanie zakładek ... 104
Wykorzystanie pozycji listy To-Do jako zakładek ... 105
Nawigacja w klasie ... 106
Przeglądanie kodu ... 107
Uzupełnianie klas ... 108
Przeszukiwanie przyrostowe ... 110
Wyszukiwanie odpowiadających znaków ograniczających ... 111
Operacje na kolumnach tekstu ... 112
Code Insight ... 112
Uzupełnianie kodu ... 113
Parametry kodu ... 115
Wyznaczanie wartości wyrażenia ... 115
Podgląd symbolu ... 116
Szablony kodu ... 116
Wiązania klawiszy edytora ... 119
Deklaracja klasy wiązania klawiszy ... 120
Implementacja klasy wiązania klawiszy ... 121
Deklaracja i implementacja procedury Register ... 126
Tworzenie i instalacja nowego pakietu fazy projektowania ... 127

Rozdział 5. Usuwanie błędów z aplikacji Kyliksa ... 131
Zintegrowany debugger ... 131
Wyznaczanie wartości wyrażeń ... 132
Okna debuggera ... 133
Menu Run ... 139
Wyłączanie debuggera ... 140
Ignorowanie wyjątków przez debugger ... 142
Ignorowanie określonych typów wyjątków przez debugger ... 142
Punkty kontrolne ... 144
Punkty kontrolne kodu źródłowego ... 144
Pozostałe typy punktów kontrolnych ... 151
Zapamiętywanie punktów kontrolnych między sesjami Kyliksa ... 154

Część II Aplikacje bazodanowe 155

Rozdział 6. Aplikacje bazodanowe ... 157
Podstawy baz danych ... 158
Bazy danych i tabele ... 158
Bazy danych i SQL ... 160
Inne zagadnienia związane z bazami danych ... 160
O bazie danych używanej w tej książce ... 161
Tworzenie bazy danych i tabel ... 162
Przegląd sposobów budowania baz danych w Kyliksie ... 162
Typy aplikacji bazodanowych obsługiwanych w Kyliksie ... 163
Komponenty używane w aplikacjach bazodanowych ... 172
Kontrolki bazodanowe ... 173
Komponenty dostępu do danych ... 174
Komponenty dbExpress ... 176
Przegląd dbExpress ... 178
Konfiguracja dbExpress ... 179
Tworzenie prostej aplikacji bazodanowej ... 185

Rozdział 7. Kontrolki bazodanowe ... 191
Konfiguracja kontrolek bazodanowych ... 192
„Kolejność tabulacji" (Tab Order) i kontrolki wizualne ... 193
Zmiana „kolejności tabulacji" ... 194
Konfiguracja kontrolek bazodanowych Kyliksa ... 195
Przygotowanie szablonu komponentów danych ... 196
Zastosowanie komponentu DBGrid ... 197
Zastosowanie komponentu DBNavigator ... 219
Zastosowanie komponentu DBText ... 221
Zastosowanie komponentu DBEdit ... 222
Zastosowania komponentu DBMemo ... 225
Zastosowanie komponentu DBListBox ... 226
Zastosowanie komponentu DBComboBox ... 229
Zastosowanie komponentu DBCheckBox ... 230
Zastosowanie komponentu DBRadioGroup ... 231
Zastosowanie komponentów DBLookupListBox i DBLookupComboBox ... 233

Rozdział 8. Klasa TField ... 237
Przegląd pól ... 238
Dostęp do pól ... 238
Pola stałe ... 243
Tworzenie pól stałych ... 244
Konfiguracja pól stałych ... 245
Praca z polami ... 253
Odczyt i zapis pól zestawu danych ... 254
Obsługa zdarzenia OnValidate ... 257
Dostęp do pola — wydajność i zarządzanie ... 258
Tworzenie nowych pól stałych ... 262
Tworzenie pola obliczeniowego ... 263
Tworzenie pól wyszukiwania ... 265
Definiowanie pól agregacji ... 267

Rozdział 9. Jednokierunkowe zestawy danych ... 271
Jednokierunkowe zestawy danych ... 272
Połączenie jednokierunkowych zestawów danych z zestawami danych typu in-memory ... 272

Zastosowanie jednokierunkowych zestawów danych 273
Konfigurowanie komponentu SQLConnection 273
Jednokierunkowe zestawy danych zwracające zestawy rekordów 274
Jednokierunkowe zestawy danych niezwracające wynikowych zestawów 278
Zastosowanie zapytań parametryzowanych 279
Tworzenie powiązań master-detail 282
Tworzenie powiązań master-detail z zapytaniami dołączanymi 284
Przygotowanie zapytań jednokierunkowych 286
Wykonywanie procedur zapamiętanych 288
Procedury zapamiętane zwracające pojedyncze wartości 289
Procedury zapamiętane zwracające zestawy danych 291

Rozdział 10. Zestawy danych typu in-memory 295
Zestawy danych typu in-memory 296
Tworzenie tabel i indeksów typu in-memory 296
Definiowanie struktury komponentu ClientDataSet po uruchomieniu aplikacji 303
Zapis danych i anulowanie zmian 305
Zastosowanie indeksów do sortowania 307
Zastosowanie zakresów 312
Zastosowanie metody ApplyRange 314
Filtrowanie 315
Filtrowanie przy użyciu właściwości 316
Właściwość FilterOptions 317
Obsługa zdarzenia OnFilterRecord 318
Zastosowanie filtra do nawigacji po danych 319
Wyszukiwanie danych 319
Zastosowanie metod FindKey oraz FindNearest 320
Zastosowanie metod GotoKey oraz GotoNearest 322
Zastosowanie metod Locate oraz Lookup 322
Walidacja na poziomie rekordu 328
Wykorzystanie zdarzenia BeforePost 328

Rozdział 11. Zaawansowane techniki bazodanowe 331
Sztuka aktualizacji danych 331
Omówienie edycji w zestawach danych typu in-memory 332
Filtrowanie oparte na statusie rekordu 338
Określanie statusu rekordu 341
Usuwanie zmian z rejestru zmian 341
Odświeżanie rekordów 345
Dostosowywanie operacji aktualizacji 348
Kontrola operacji aktualizacji przy użyciu właściwości komponentu DataSetProvider 348
Programowa obsługa operacji aktualizacji 353
Obsługa błędów operacji aktualizacji 358
Dodatkowe techniki 361
Zastosowanie komponentu SQLMonitor 361
Tworzenie kopii kursora 362
Aktualizowanie tylko jednego rekordu 363

Rozdział 12. Tworzenie sterowników dbExpress 367
Biblioteki producentów serwerów bazodanowych 368
Uruchomienie środowiska 369
Połączenie z serwerem bazodanowym 369
Konfiguracja uchwytu polecenia 369
Przygotowanie polecenia SQL 370
Przekazywanie parametrów startowych 370

Wykonywanie polecenia SQL 371
Wiązanie bufora rekordu 371
Ładowanie rekordów 372
Zwalnianie uchwytów oraz rozłączanie....373
Implementacja podstawowych klas dbExpress....373
Klasa SQLDriver 374
Klasa SQLConnection 376
Klasa SQLCommand 382
Klasa SQLCursor....398
Klasa SQLMetaData....403
Pliki źródłowe interfejsu dbExpress 403

Część III Zaawansowane możliwości Kyliksa 405

Rozdział 13. Aplikacje wielowątkowe 407
Zalety wielowątkowości 409
Tworzenie aplikacji wielowątkowych 411
Używanie klasy TThread....411
Synchronizacja wątków 425
Używanie metody Synchronize 425
Praca z sekcjami krytycznymi 426
Czekanie na wątek 429
Używanie klasy TEvent....433
Blokowanie obiektów 434
Inne techniki programowania wielowątkowego 435
Zmienne lokalne wątku....435
Testowanie wątków 436
Wielowątkowy dostęp do baz danych 437

Rozdział 14. Biblioteki współdzielone....439
Omówienie bibliotek współdzielonych....440
Budowanie przykładowej biblioteki współdzielonej 441
Pisanie funkcji eksportowanych 443
Kontrola nazwy biblioteki współdzielonej 444
Ładowanie procedur z bibliotek współdzielonych....449
Umiejscawianie bibliotek współdzielonych 451
Tworzenie modułów importu dla bibliotek współdzielonych....453
Dynamiczne ładowanie bibliotek współdzielonych....454
Deklarowanie zmiennych 456
Dynamiczne ładowanie bibliotek współdzielonych 457
Uzyskiwanie adresu funkcji lub procedury 457
Zwalnianie biblioteki współdzielonej....458
Testowanie bibliotek współdzielonych....460
Testowanie przy użyciu aplikacji macierzystej 460
Testowanie przy użyciu grupy projektu 462
Sekcje initialization i finalization biblioteki współdzielonej 463
Definiowanie kodu inicjującego....463
Definiowanie procedury kończącej działanie....463

Rozdział 15. Tworzenie komponentów....465
Przegląd obiektów....466
Od rekordu do klasy....466
Hermetyzacja i widoczność składowych....470
Definiowanie interfejsu fazy wykonania....471
Dziedziczenie i polimorfizm....474

Wprowadzenie do tworzenia komponentów 477
Wskazówki dotyczące projektowania komponentów 478
Przykładowy prosty komponent: definiowanie nowych wartości domyślnych właściwości 479
Wykorzystanie kreatora Component 480
Pokrywanie metody 482
Implementacja pokrywanego konstruktora 483
Testowanie nowego komponentu 485
Instalowanie komponentu 487
Tworzenie pakietu fazy projektowania 488
Konfigurowanie pakietu 490
Przykład z właściwościami 491
Definiowanie pól obiektu 492
Definiowanie właściwości 493
Definiowanie metod 493
Pokrywanie istniejących metod 495
Implementacja metod pokrywanych 496
Tworzenie zdarzeń 497
Końcowe poprawki komponentu: zwracanie uwagi na szczegóły 499
Zagadnienie dodatkowe: zwiększanie widoczności właściwości 506
Przykład zwiększania widoczności właściwości 508

Rozdział 16. Wykorzystanie interfejsów 511
Przegląd interfejsów 512
Argumenty za interfejsami 514
Deklarowanie interfejsu 516
Implementowanie interfejsów 518
Interfejsy i rozdzielanie nazw metod 526
Implementowanie interfejsu przez delegowanie 527
Uwagi na temat implementowania interfejsu przez delegowanie 529
Przykład interfejsu — dostarczyciel danych 530

Część IV Technologie internetowe 537

Rozdział 17. Przegląd technologii internetowych 539
Protokoły, technologie i pojęcia 540
Dokumenty RFC 540
Adresy IP 540
Nazwy domen 541
TCP/IP i UDP 542
Gniazda i porty 543
SGML 543
HTML 544
FTP 545
HTTP 545
MIME 546
World Wide Web 546
Serwery WWW 547
Przeglądarki internetowe 547
Apache 547
CGI i biblioteki DSO 548
Server Side Includes (SSI) 549
Przegląd rozszerzeń serwerów WWW 550
Krótkie omówienie interakcji w sieci WWW 551
Części adresu URL 552
Typy wywołań 554

Praca z HTML 555
Przegląd HTML 556
Wysyłanie danych do rozszerzeń serwera WWW przy użyciu HTML 557
Znacznik obrazu 558
Znacznik zakotwiczenia 559
Formularze HTML 560
Kompilacja serwera Apache wykorzystującego biblioteki DSO 565

Rozdział 18. Tworzenie rozszerzeń serwera Apache przy użyciu technologii WebBroker 569
Tworzenie prostego rozszerzenia CGI 570
Definiowanie akcji 571
Instalacja i użytkowanie rozszerzenia CGI 575
Dodawanie zmiennej środowiskowej LD_LIBRARY_PATH 576
Zapisywanie aplikacji CGI do katalogu ScriptAlias 578
Uruchamianie aplikacji CGI za pośrednictwem przeglądarki 579
Tworzenie i konfiguracja prostego rozszerzenia DSO 581
Tworzenie projektu DSO 581
Producenci 583
Instalacja i użytkowanie biblioteki DSO 587
Zapisywanie biblioteki DSO do katalogu serwera Apache 587
Dodawanie DSO do pliku httpd.conf 588
Zatrzymywanie i uruchamianie serwera Apache 589
Uruchamianie biblioteki DSO za pośrednictwem przeglądarki 589

Rozdział 19. Zaawansowane zagadnienia dotyczące Web Broker 593
Pozyskiwanie danych z formularzy 594
Tworzenie internetowej aplikacji bazodanowej 598
Rozszerzenia serwera WWW i współbieżność 598
Obiekty dostarczające dane 600
Formatowanie komórek komponentów TableProducer 605
Cookies i obiekt WebRequest 606
Pobieranie i ustawianie cookies 607
Wykorzystanie cookies i przekierowania 609
Zawartość obiektu WebRequest 612
Usuwanie błędów z rozszerzeń serwera WWW 615
Konwersja projektu CGI do projektu DSO 615
Usuwanie błędów z projektu DSO 616

Rozdział 20. Przegląd Internet Direct 621
Internet Direct 621
Komponenty Internet Direct 624
Komponenty klientów Indy 625
Komponenty serwerów Indy 628
Pozostałe komponenty Indy 631
Pobieranie uaktualnionych komponentów Internet Direct 633
Wykorzystanie komponentów Internet Direct 634
Działanie klientów Internet Direct 634
Wykorzystanie TIdAntiFreeze 636
Działanie serwerów Internet Direct 636
Wykorzystanie menedżera wątków 638
Licencja Internet Direct 638
Zmodyfikowana licencja BSD dla Indy 639
Licencja Indy MPL (Mozilla Public License) 639
Spełnianie wymagań licencji Indy w aplikacjach Kyliksa 640
Pomoc techniczna 640

Rozdział 21. Wykorzystanie Internet Direct 643
Co było pierwsze: klient czy serwer? 643
Wykorzystanie Telnetu do testowania serwera tekstowego 644
Przykład prostego serwera 645
TIdTCPServer i wątki 648
Wywołania blokujące i współbieżność 649
OnExecute i wyjątki 650
Przykład serwera bazy danych 651
Tworzenie serwera bazy danych 651
Tworzenie klienta bazy danych 654
Obsługa wyjątków w aplikacjach klienckich 656
Rozpoznawanie zamknięcia połączenia przez klienta 657
Testowanie serwera za pomocą wielowątkowego klienta 657
Wysyłanie poczty elektronicznej za pomocą TIdSMTP 661
Tworzenie wiadomości 662
Tworzenie klienta SMTP 663
Tworzenie klienta w wątku 663
Tworzenie egzemplarza wątku klienta 665
Uaktualnianie interfejsu użytkownika przez wątek 666
Serwer i klient kodów pocztowych 668
Definiowanie protokołu kodów pocztowych 669
Serwer kodów pocztowych 670
Klient kodów pocztowych 672
Tworzenie serwera konsoli 675
Tworzenie przykładowego serwera konsoli 675
Testowanie tekstowego serwera konsoli 677

Skorowidz 679

Rozdział 8.
Klasa TField

W tym rozdziale:

- omówienie pól,
- pola stałe,
- praca z polami w czasie wykonania,
- tworzenie nowych pól stałych.

Zestaw danych służy do działania z jednym lub kilkoma rekordami wynikowego zestawu, ale nie pozwala na kontrolowanie zachowań w przypadku pojedynczych pól tych rekordów. Do pracy z polami stosuje się obiekty pól, które są instancjami klas pochodnych klasy `TField`. Każda z tych klas nich obsługuje określony typ pola. Zależnie od wynikowego zestawu, z którym pracujemy, będziemy wykorzystywać egzemplarze różnych klas, takich jak `TStringField`, `TIntegerField`, `TFloatField`, `TMemoField`, `TBlobField`, `TDateTimeField i inne`.

W rozdziale tym omówimy stosowanie pól. Na początku dokonamy ogólnego przeglądu pól, włączając w to informacje na temat dostępu do pól dynamicznych w czasie wykonania. W dalszej części zostaną przybliżone pola stałe stosowane w fazie projektowania do konfiguracji obiektów pól. Zostaną omówione następujące zagadnienia: tworzenie pól obliczeniowych, zastosowanie pól wyszukiwania, definiowanie pól agregacji, użycie walidacji oraz określenie ograniczeń na poziomie pola.

W rozdziale znajduje się kilka przykładów, które prezentują różne techniki związane z polami. W celu uproszczenia tych przykładów zostanie użyty szablon *GetSales*, który został utworzony w poprzednim rozdziale. Jeżeli nie masz tego szablonu to do poprawnego wykonania przykładów zawartych w tym rozdziale wymagane będzie wykonanie kroków opisanych w akapicie rozdziału 7. — „Przygotowanie szablonu komponentów danych”.

Przegląd pól

Pola są jedną z najważniejszych kategorii obiektów w aplikacjach bazodanowych. Umożliwiają odczytywanie pojedynczych wartości pól zestawu rekordów udostępnianych przez zestawy danych. Jeśli zestaw danych pozwala na modyfikację danych, które zawiera, wtedy obiekty pól służą do przypisywania nowych wartości poszczególnym polom (zestaw danych posiada metody pozwalające na przypisywanie wartości wielu polom w pojedynczym zapytaniu).

Obiekty pól pozwalają także kontrolować sposób, w jaki poszczególne pola są wyświetlane w kontrolkach bazodanowych. Na przykład za pomocą pól można zdefiniować maski edycji, które określają format i zakres danych wprowadzanych do powiązanej kontrolki bazodanowej. Można również utworzyć ograniczenia i procedury obsługi zdarzeń, które dokonują walidacji danych wprowadzanych przez użytkownika.

Domyślnie po uruchomieniu zestaw danych tworzy jeden obiekt pola dla każdego powiązanego z nim pola. Pola tworzone w ten sposób są określane mianem *pól dynamicznych*. Po uruchomieniu obiekty te udostępniane są przez właściwość *Fields* zestawu danych lub przy użyciu metody `FieldByName`. Ewentualnie można utworzyć obiekt pola w fazie projektowania.

Tak utworzony obiekt pola określany jest mianem pola stałego. Pozwala w dość prosty sposób w fazie projektowania konfigurować pole. Podobnie jak w przypadku pola dynamicznego, pole stałe jest w fazie wykonania dostępne przez właściwość *Fields* lub przy użyciu metody `FieldByName`.

Dostęp do pól

Jak wspomniano wcześniej, do dostępu do obiektów pól służy właściwość *Fields* lub metoda `FieldByName` zestawu danych. Typem danych właściwości *Fields* jest klasa `TFields`, która może przechowywać odwołanie do kilku pól. Poniżej znajduje się deklaracja takiej klasy.

```
TFields = class(TObject)
private
 FList: TList;
 FDataSet: TDataSet;
 FSparseFields: Integer;
 FOnChange: TNotifyEvent;
 FValidFieldKinds: TFieldKinds;
protected
 procedure Changed;
 procedure CheckFieldKind(FieldKind: TFieldKind; Field: TField);
 function GetCount: Integer;
 function GetField(Index: Integer): TField;
 procedure SetField(Index: Integer; Value: TField);
 procedure SetFieldIndex(Field: TField; Value: Integer);
 property SparseFields: Integer read FSparseFields write FSparseFields;
 property OnChange: TNotifyEvent read FOnChange write FOnChange;
 property ValidFieldKinds: TFieldKinds read FValidFieldKinds
  write FValidFieldKinds;
```

```
public
 constructor Create(ADataSet: TDataSet);
 destructor Destroy; override;
 procedure Add(Field: TField);
 procedure CheckFieldName(const FieldName: string);
 procedure CheckFieldNames(const FieldNames: string);
 procedure Clear;
 function FindField(const FieldName: string): TField;
 function FieldByName(const FieldName: string): TField;
 function FieldByNumber(FieldNo: Integer): TField;
 procedure GetFieldNames(List: TStrings);
 function IndexOf(Field: TField): Integer;
 procedure Remove(Field: TField);
 property Count: Integer read GetCount;
 property DataSet: TDataSet read FDataSet;
 property Fields[Index: Integer]: TField read GetField
  write SetField; default;
end;
```

Należy zauważyć, że właściwość *Fields* w deklaracji klasy `TFields` jest zarówno właściwością domyślną, jak i indeksowaną (klasa może mieć tylko jedną właściwość domyślną). Przy odwołaniu do właściwości *Fields* egzemplarza klasy `TFields` należy podać liczbę całkowitą, która identyfikuje jeden z obiektów pól zestawu danych. Jeżeli właściwość indeksowana jest domyślna to przy odwołaniu do obiektu danej klasy następuje odwołanie do tej właściwości. Oznacza to, że właściwość *Fields* zestawu danych jest stosowana do bezpośredniego dostępu do właściwości *Fields* instancji klasy `TFields`. Może się to wydać trochę niezrozumiałe, ale oznacza tylko, że poniższe dwa odniesienia funkcjonalnie są identyczne.

```
ClientDataSet1.Fields.Fields[0];

ClientDataSet1.Fields[0];
```

Ponieważ drugie odwołanie jest prostsze, preferuje je większość programistów.

Liczba całkowita stosowana jako odwołanie do określonego pola we właściwości *Fields* odnosi się do położenia pola w zestawie danych. W przypadku stosowania domyślnych obiektów pól tworzonych przez zestaw danych, położenie to jest jednakowe z położeniem pola w powiązanym zestawie wynikowym. Jeśli stosuje się obiekty pól stałych, wtedy jest to odwołanie do położenia pola w edytorze pól.

Metoda `FieldByName` zestawu danych pobiera łańcuch i zwraca odwołanie do pola zestawu danych, którego nazwa jest zgodna z łańcuchem (w nazwie nie jest rozróżniana wielkość liter). Jeśli żadna nazwa pola nie zgadza się z łańcuchem, wtedy wywoływany jest wyjątek.

Przyjmując, że stosowane są domyślne obiekty pól i pierwsze pole zestawu danych *ClientDataSet1* ma nazwę *CUST_NO*, poniższe dwa odwołania odnoszą się do dokładnie tego samego obiektu pola.

```
ClientDataSet1.Fields[0];

ClientDataSet1.FieldByName('CUST_NO');
```

Odwołanie do pola stosowane jest w celu uzyskania dostępu do właściwości i wywołania metod określonej klasy pola. Ponieważ wszystkie obiekty pól są pochodnymi klasy `TField`, dziedziczą one jej interfejs. Poniżej zawarto częściowy wydruk klasy `TField`, przedstawiający tylko jej sekcje `public` oraz `published` (odpowiednio elementy fazy wykonania oraz projektowania).

```
TField = class(TComponent)
// deklaracje private oraz protected nie zostały pokazane
public
 constructor Create(AOwner: TComponent); override;
 destructor Destroy; override;
 procedure Assign(Source: TPersistent); override;
 procedure AssignValue(const Value: TVarRec);
 procedure Clear; virtual;
 procedure FocusControl;
 function GetData(Buffer: Pointer; NativeFormat: Boolean = True): Boolean;
 function GetParentComponent: TComponent; override;
 function HasParent: Boolean; override;
 class function IsBiob: Boolean; virtual;
 function IsValidChar(InputChar: Char): Boolean; virtual;
 procedure RefreshLookupList;
 procedure SetData(Buffer: Pointer; NativeFormat: Boolean = True);
 procedure SetFieldType(Value: TFieldType); virtual;
 procedure Validate(Buffer: Pointer);
 property AsBCD: TBcd read GetAsBCD write SetAsBCD;
 property AsBoolean: Boolean read GetAsBoolean write SetAsBoolean;
 property AsCurrency: Currency read GetAsCurrency write SetAsCurrency;
 property AsDateTime: TDateTime read GetAsDateTime write SetAsDateTime;
 property AsSQLTimeStamp: TSQLTimeStamp read GetAsSQLTimeStamp
  write SetAsSQLTimeStamp;
 property AsFloat: Double read GetAsFloat write SetAsFloat;
 property AsInteger: Longint read GetAsInteger write SetAsInteger;
 property AsString: string read GetAsStrinq write SetAsString;
 property AsVariant: Variant read GetAsVariant write SetAsVariant;
 property AttributeSet: string read FAttributeSet write FAttributeSet;
 property Calculated: Boolean read GetCalculated
  write SetCalculated default False;
 property CanModify: Boolean read GetCanModify;
 property CurValue: Variant read GetCurValue;
 property DataSet: TDataSet read FDataSet write SetDataSet stored False;
 property DataSize: Integer read GetDataSize;
 property DataType: TFieldType read FDataType;
 property DisplayName: string read GetDisplayName;
 property DisplayText: string read GetDisplayText;
 property EditMask: string read FEditMask write SetEditMask;
 property EditMaskPtr: string read FEditMask;
 property FieldNo: Integer read GetFieldNo;
 property FullName: string read GetFullName;
 property IsIndexField: Boolean read GetIsIndexField;
 property IsNull: Boolean read GetIsNull;
 property Lookup: Boolean read GetLookup write SetLookup;
 property LookupList: TLookupList read GetLookupList;
 property NewValue: Variant read GetNewValue write SetNewValue;
 property Offset: Integer read FOffset;
 property OldValue: Variant read GetOldValue;
 property ParentField: TObjectField read FParentField
  write SetParentField;
```

```
  property Size: Integer read GetSize write SetSize;
  property Text: string read GetEditText write SetEditText;
  property ValidChars: TFieldChars read FValidChars write FValidChars;
  property Value: Variant read GetAsVariant write SetAsVariant;
 published
  property Alignment: TAlignment read FAlignment
   write SetAlignment default taLeftJustify;
  property AutoGenerateValue: TAutoRefreshFlag read FAutoGenerateValue
   write SetAutoGenerateValue default arNone;
  property CustomConstraint: string read FCustomConstraint
   write FCustomConstraint;
  property ConstraintErrorMessage: string read FConstraintErrorMessage
   write FConstraintErrorMessage;
  property DefaultExpression: string read FDefaultExpression
   write FDefaultExpression;
  property DisplayLabel: string read GetDisplayLabel write SetDisplayLabel
   stored IsDisplayLabelStored;
  property DisplayWidth: Integer read GetDisplayWidth
   write SetDisplayWidth
   stored IsDisplayWidthStored;
  property FieldKind: TFieldKind read FFieldKind
   write SetFieldKind default fkData;
  property FieldName: string read FFieldName write SetFieldName;
  property HasConstraints: Boolean read GetHasConstraints default False;
  property Index: Integer read GetIndex write SetIndex stored False;
  property ImportedConstraint: string read FImportedConstraint
   write FImportedConstraint;
  property LookupDataSet: TDataSet read FLookupDataSet
   write SetLookupDataSet;
  property LookupKeyFields: string read FLookupKeyFields
   write SetLookupKeyFields;
  property LookupResultField: string read FLookupResultField
   write SetLookupResultField;
  property KeyFields: string read FKeyFields write SetKeyFields;
  property LookupCache: Boolean read FLookupCache
   write SetLookupCache default False;
  property origin: string read FOrigin write FOrigin;
  property ProviderFlags: TProviderFlags read FProviderFlags
   write FProviderFlags default [pfInWhere, pfInUpdate];
  property ReadOnly: Boolean read FReadOnly
   write SetReadOnly default False;
  property Required: Boolean read FRequired write FRequired default False;
  property Visible: Boolean read FVisible write SetVisible default True;
  property OnChange: TFieldNotifyEvent read FOnChange write FOnChange;
  property OnGetText: TFieldGetTextEvent read FOnGetText write FOnGetText;
  property OnSetText: TFieldSetTextEvent read FOnSetText write FOnSetText;
  property OnValidate: TFieldNotifyEvent read FOnValidate
   write FOnValidate;
 end;
```

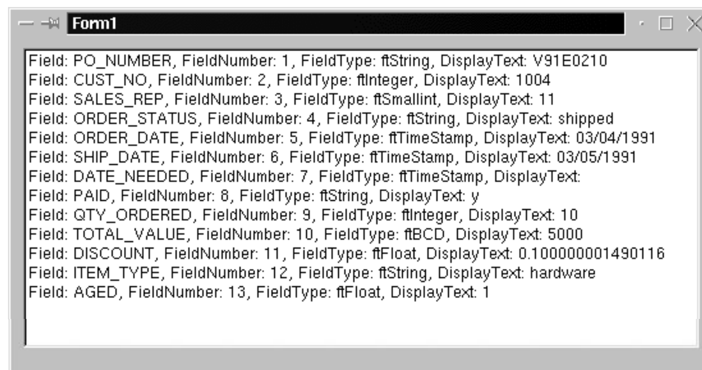
Jak można zauważyć, klasa TField zawiera mnóstwo informacji na temat każdego pola. Ponadto poszczególne pochodne tej klasy posiadają dodatkowe metody i właściwości, które mają specyficzne dla nich zastosowanie. Na przykład klasa TBlobField zawiera metody obsługujące ładowanie danych z pliku lub ze strumienia.

Poniższy przykład pokazuje, jak uzyskać dostęp do obiektów pól zestawu danych.

1. Utwórz nowy projekt.
2. Do głównego formularza dodaj komponent *Memo* z zakładki *Standard* palety komponentów. Należy również dodać komponent szablonu *GetSales* z zakładki *Templates*, który został utworzony w rozdziale 7.
3. Dodaj moduł `TypInfo` do sekcji `uses` głównego formularza. Moduł ten pozwala na dostęp do informacji na temat działającej aplikacji (*runtime-type information* — *RTTI*). *RTTI* jest informacją typu tekstowego dotyczącą deklaracji typów i jest wykorzystany w procedurze obsługi zdarzenia dodanej w następnym kroku przy konwersji typu wyliczeniowego na łańcuchowy.
4. Dodaj do głównego formularza procedurę obsługi zdarzenia *OnCreate*. Wprowadź do niej poniższy kod:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  i: Integer;
begin
  Memo1.Lines.Clear;
  for i := 0 to SQLClientDataSet1.Fields.Count -1 do
  begin
    Memo1.Lines.Add('Field: ' +
      SQLClientDataSet1.Fields[i].FieldName +
      ', FieldNumber: ' +
      IntToStr(SQLClientDataSet1.Fields[i].FieldNo) +
      ', FieldType: ' +
      GetEnumName(TypeInfo(TFieldType),
      Ord(SQLClientDataSet1.Fields[i].DataType)) +
      ', DisplayText: ' + SQLClientDataSet1.Fields[i].DisplayText);
  end;
end;
```

5. Uruchom projekt. Wyświetlony formularz powinien wyglądać podobnie jak na rysunku 8.1.



Rysunek 8.1. *Zastosowanie właściwości Fields zestawu danych do dostępu do poszczególnych obiektów pól*

Kod źródłowy tego projektu znajduje się w katalogu *fields*.

Pola stałe

Właściwość *Fields* zestawu danych to właściwość `publiczna`, co oznacza, że jest elementem interfejsu czasu wykonania i dostępna jest dopiero po uruchomieniu. Jeśli musimy konfigurować pola w czasie działania aplikacji, należy utworzyć pola stałe. Pola takie są tworzone w edytorze pól.

Można wyróżnić dwa typy pól stałych: pola oparte na polach zestawu danych oraz pola, które na nich nie bazują. Pola stałe oparte na polach zestawu danych dostarczają odpowiedników pól dynamicznych w fazie projektowania. Pola niebazujące na polach zestawu danych nie posiadają odpowiednika w postaci pól dynamicznych. Do pól tych zaliczają się: pola obliczeniowe, wyszukiwania oraz agregacji.

Niestety dokumentacja Kyliksa nie precyzuje, jak nazywać takie specjalne pola stałe. Pola te są określane mianem *nowych pól stałych*. Chociaż termin nie jest zbyt jasny, jest zgodny z pozycją menu służącą do tworzenia takich pól. Aby utworzyć jedno takie specjalne pole stałe, należy kliknąć prawym klawiszem myszy edytor pól i wybrać *New Field*, a następnie zdefiniować pole w oknie dialogowym *New Field*. Ostatecznie w tej książce specjalne pola stałe będą określane terminem *nowych pól stałych*, natomiast pozostałe typy pól stałych — *polami stałymi*.

Pola stałe nie mogą występować w zestawie danych razem z polami dynamicznymi. Nawet jeśli tworzy się tylko jedno pole stałe dowolnego typu, zestaw danych nie wygeneruje pól dynamicznych i jedynymi polami, do których będzie istniał dostęp, będą pola stałe. Jeżeli mamy komponent *SQLClientDataSet* używający zapytania `select` do pobrania dziesięciu pól tabeli i utworzymy tylko jedno pole stałe to tylko pole powiązane z polem stałym może być przeglądane, edytowane lub na inny sposób udostępnione.

W rezultacie ma to znaczący wpływ na stosowanie zestawów danych. Ściśle mówiąc, w przypadku tworzenia pól stałych zmiany w strukturze powiązanej tabeli narzucą konieczność dokonania również zmian w tych polach. W niektórych sytuacjach konieczne będzie usunięcie wszystkich pól stałych i ponowne ich dodanie po dokonaniu zmian w strukturze tabeli lub pól zwracanych przez zapytanie. Jeśli zdefiniowano właściwości i procedury obsługi zdarzeń dla usuniętych pól stałych, należy ponownie skonfigurować nowo dodane pola.

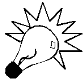
Nie trzeba dodawać, że należy dążyć do tego, aby przy tworzeniu pól stałych, na ile to jest możliwe, uniknąć dokonywania zmian struktury powiązanych tabel.

Jednym ze sposobów realizacji tego zalecenia jest tworzenie aplikacji przy użyciu pól dynamicznych i następnie dodanie pól stałych, ale tylko po upewnieniu się, że struktury danych są prawidłowo zaprojektowane. Niestety nieprzewidziane zmiany w strukturze danych są czasami konieczne i wtedy wiąże się to również z dodatkowym nakładem pracy przy polach stałych. Więcej na ten temat w dalszej części rozdziału.

Tworzenie pól stałych

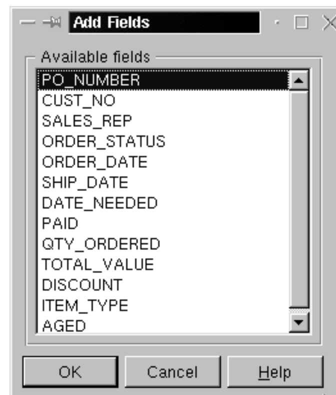
Jak już wspominaliśmy wcześniej, pola stałe tworzone są w edytorze pól. Aby go otworzyć, należy kliknąć dwukrotnie zestaw danych lub kliknąć prawym klawiszem myszy zestaw i wybrać edytor pól (patrz rysunek 8.2).

Rysunek 8.2.
Edytor pól



Zwykle nie dodaje się pól stałych do jednokierunkowego zestawu danych, ponieważ nie posiada on mechanizmu do zapisywania zmian w danych do bazy danych. Jedyną sytuacją, w której dodanie pól stałych do jednokierunkowego zestawu danych ma sens, jest zastosowanie ich do odczytu danych pola przy nawigacji po zestawie danych. Przykładem tego jest generowanie raportu.

Po otwarciu edytora należy za pomocą wciśnięcia klawiszy *Ctrl+A* lub kliknięcia prawym klawiszem myszy okna edytora i wybrania *Add Fields* dodać jedno lub kilka pól stałych. Kylix wyświetli okno dialogowe *Add Fields*. W oknie tym trzeba zaznaczyć jedno lub kilka pól zestawu danych, dla którego mają zostać utworzone pola stałe, a następnie kliknąć przycisk *OK* (patrz rysunek 8.3).



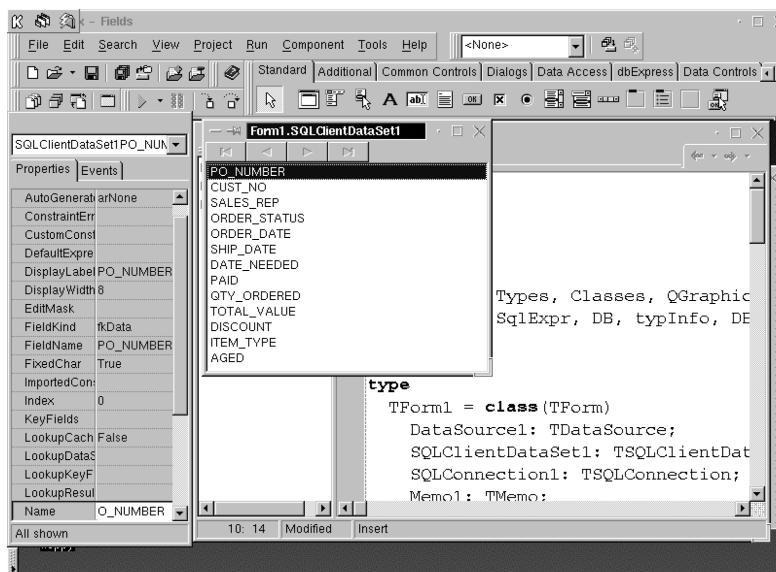
Rysunek 8.3.
Okno Add Fields

Aby w jednej operacji w edytorze pól dodać pola stałe dla każdego pola powiązanego zestawu danych, należy wcisnąć klawisze *Ctrl+F* lub kliknąć prawym klawiszem myszy okno edytora, a następnie wybrać opcję *Add All Fields*. W celu dodania nowego pola stałego należy wcisnąć klawisze *Ctrl+N* lub kliknąć prawym klawiszem myszy okno edytora, a następnie wybrać opcję *New Fields*. Operacja dodania nowego pola stałego została opisana w dalszej części rozdziału.

> Aby umożliwić zarządzanie polami stałymi, edytor pól posiada również prosty nawigator po zestawie danych stosowany w fazie projektowania. Za pomocą czterech przycisków nawigacji znajdujących się w górnej części edytora pól można przechodzić do pierwszego, następnego, poprzedniego i ostatniego rekordu zestawu danych, który jest wyświetlony w edytorze. Może to być przydatne w fazie projektowania, gdy otwarty jest zestaw danych i chcemy zobaczyć dane w dowolnej kontrolce bazodanowej dodanej do interfejsu użytkownika.

Konfiguracja pól stałych

Pola stałe konfigurowane są w fazie projektowania przy użyciu okna *Object Inspector*. Aby rozpocząć konfigurację, należy wybrać pole w edytorze pól lub z listy obiektów okna *Object Inspector*. Rysunek 8.4 przedstawia okno *Object Inspector* z zaznaczonym egzemplarzem klasy `TStringField`.



Rysunek 8.4. *Po zaznaczeniu pola stałego w edytorze pól jego właściwości mogą być ustawiane w oknie Object Inspector*

Wielu właściwości pola stałego używają inne obiekty, które z nim współpracują. Na przykład domyślne obiekty kolumn tworzone przez komponent *DBGrid* stosują właściwość *DisplayLabel* pola stałego do definiowania tekstu nagłówka kolumny. Podobnie właściwości *Alignment* oraz *DisplayFormat* wpływają na zawartość pola wyświetlanego w kontrolkach bazodanowych.

Poniżej zostały opisane niektóre właściwości, które są ustawiane w fazie projektowania. W celu prezentacji zostanie utworzony prosty projekt, który zostanie również wykorzystany w dalszej części rozdziału do pokazania innych technik i funkcjonalności.

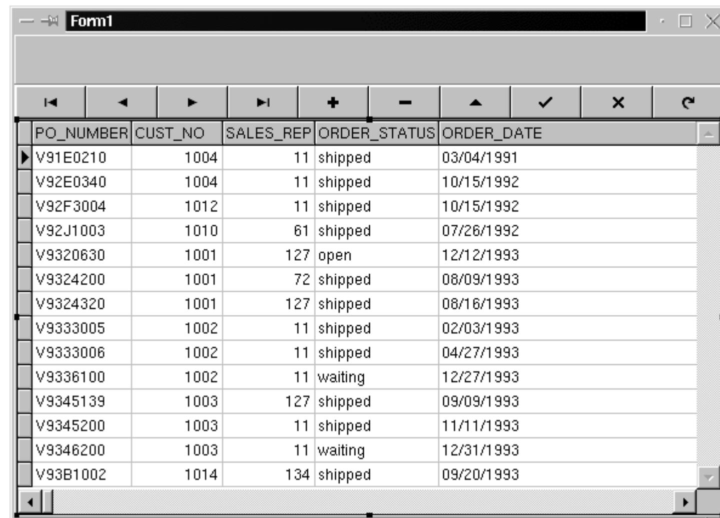
1. Utwórz nowy projekt.
2. Wybierz z menu *File/New*, a następnie kreator *Data Module* z zakładki *New* okna *Object Repository*. Kliknij *OK*.

3. Dodaj do modułu danych szablon komponentu (utworzony w rozdziale 7.) *GetSales* z zakładki *Templates* palety komponentów.

4. Dodaj moduł danych do głównego formularza, wracając do niego i wybierając z menu *File/Use Unit*, a następnie *Unit2* w oknie dialogowym *Use Unit*.

5. Dodaj do głównego formularza dwa panele. Ustaw dla właściwości *Caption* obydwu paneli pusty łańcuch. Następnie ustaw wartość właściwości *Align* jednego panelu na *alTop*, a drugiego na *alClient*.

6. Dodaj komponenty *DBNavigator* oraz *DBGrid* z zakładki *Data Controls* palety komponentów do panelu, którego właściwość *Align* ma wartość *alClient*. Następnie ustaw wartość właściwości *Align* komponentu *DBNavigator* na *alTop*, a komponentu *DBGrid* na *alClient*. Ustaw wartość właściwości *DataSource* obydwu kontrolek bazodanowych na *DataModule2.DataSource1*. Formularz powinien wyglądać podobnie jak na rysunku 8.5.



PO_NUMBER	CUST_NO	SALES_REP	ORDER_STATUS	ORDER_DATE
V91E0210	1004	11	shipped	03/04/1991
V92E0340	1004	11	shipped	10/15/1992
V92F3004	1012	11	shipped	10/15/1992
V92J1003	1010	61	shipped	07/26/1992
V9320630	1001	127	open	12/12/1993
V9324200	1001	72	shipped	08/09/1993
V9324320	1001	127	shipped	08/16/1993
V9333005	1002	11	shipped	02/03/1993
V9333006	1002	11	shipped	04/27/1993
V9336100	1002	11	waiting	12/27/1993
V9345139	1003	127	shipped	09/09/1993
V9345200	1003	11	shipped	11/11/1993
V9346200	1003	11	waiting	12/31/1993
V93B1002	1014	134	shipped	09/20/1993

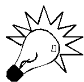
Rysunek 8.5. *Prosty projekt interfejsu użytkownika demonstrujący zastosowanie i konfigurację pola*

7. Teraz w celu wyświetlenia edytora pól kliknij dwukrotnie komponent *SQLClientDataSet* w module danych. Aby utworzyć pola stałe dla wszystkich pól komponentu *SQLClientDataSet*, należy nacisnąć klawisze *Ctrl+F*.

8. Zapisz projekt.

Kod źródłowy tego projektu znajduje się w katalogu *persistentfields*.

W dalszej części rozdziału zostaną ustawione w fazie projektowania właściwości niektórych obiektów pól stałych utworzonych w powyższym projekcie. Edytor pól może zostać otwarty, ponieważ będzie przydatny przy wybieraniu pola, które zostanie skonfigurowane w oknie *Object Inspector*. Jeśli jednak został zamknięty, wtedy w celu jego ponownego otwarcia wystarczy kliknąć dwukrotnie komponent *SQLClientDataSet*. Po otwarciu edytor pól wyświetli wszystkie pola stałe, które zostały zdefiniowane dla zestawu danych.



> Projekt ten nie zawiera kodu obsługującego zapis zmian dokonanych na danych. W celu ich zapisania w tabeli *SALES* konieczne będzie wykonanie odwołania do metody `ApplyUpdates`. Przykładowy kod realizujący to zadanie znajduje się na końcu rozdziału 6.

Ustawianie właściwości DisplayFormat

Właściwość *DisplayFormat* stosowana jest przez pola zawierające dane typu liczbowego i decyduje o tym, jak będą wyświetlane w kontrolkach bazodanowych. Właściwość ta może być użyta do określenia, ile jest wyświetlanych miejsc po przecinku, jak również czy dołączyć separator części tysięcznych.

Właściwość *DisplayFormat* może posiadać jedną, dwie lub trzy definicje formatu. W przypadku dwóch lub więcej definicji stosowany jest średnik. Po określeniu tylko jednego formatu dotyczy on wszystkich wartości numerycznych pola. W przypadku dwóch definicji formatu pierwsza z nich dotyczy wartości dodatnich oraz zera, natomiast druga tylko wartości ujemnych. Wreszcie po określeniu trzech formatów pierwszy z nich dotyczy wartości dodatnich, drugi wartości ujemnych, a trzeci stosowany jest dla wartości zerowych. W przypadku dodania trzech znaków średnika, ale bez definiowania jednego lub kilku formatów, zostanie dla nich zastosowana domyślna wartość właściwości *DisplayFormat*.

Definicja formatu wyświetlania tworzona jest przy użyciu specyfikatorów i stałych łańcuchowych. Stosowane specyfikatory są opisane w tabeli 8.1. Tabela 8.2 zawiera przykłady ich zastosowania.

Tabela 8.1. *Specyfikatory formatu wyświetlania wartości pól numerycznych*

Specyfikator	Opis
`#`	Znak # reprezentuje dowolną cyfrę. W przypadku zastosowania go do definiowania ilości miejsc po przecinku wyświetlane są tylko wartości dziesiętne różne od zera. Aby wyświetlić stałą ilość miejsc po przecinku, należy użyć wartości `0`.
`0`	Zero reprezentuje dokładnie jedną cyfrę. Aby wyświetlić stałą ilość miejsc po przecinku, należy użyć tego specyfikatora w ułamkowej części definiowanego formatu. Aby wstawić zera początkowe, należy umieścić `0` w części całkowitej definiowanego formatu.
`,`	Jeśli w definicji formatu pojawi się znak `,` , wtedy zostaną wyświetlone separatory części tysięcznej. Znakiem stosowanym do oddzielania części tysięcznej jest znak, który został zdefiniowany w systemie operacyjnym.
`.`	Znak `.` , poprzedzony przez jeden lub kilka znaków # lub `0` stosowany jest w celu dołączenia kropki rozdzielającej część dziesiętną od całkowitej. Znakiem stosowanym do oddzielania części dziesiętnej od całkowitej jest znak, który został zdefiniowany w systemie operacyjnym.
`E+`	Specyfikator ten wyświetla dane w notacji naukowej (wykładniczej). Specyfikator `E+` lub `e+` służy do wyświetlania wartości w notacji wykładniczej z uwzględnieniem znaku, natomiast `E-` lub `e-` dodaje znak tylko dla liczb ujemnych.
`'stałe łańcuchowe'`	Dowolne znaki pojawiające się wewnątrz pojedynczych lub podwójnych znaków cudzysłowu są stałymi łańcuchowymi i jako takie będą wyświetlane.

Tabela 8.2. *Przykład zastosowania specyfikatorów właściwości DisplayFormat*

Przykład	Działanie
`,#.00`	Wyświetla wszystkie cyfry, stosując separatory części tysięcznej i dokładnie dwa miejsca po przecinku. Dla wartości pomiędzy 1 a –1 nie wyświetla zera, aby wyświetlić zero dla liczb ułamkowych należy zamienić # na 0.
`000000.00`	Wyświetla wszystkie cyfry, w tym sześć dla części całkowitej a dwie dla dziesiętnej. W tym przypadku brak separatorów części tysięcznej. Liczby zawierające pięć lub mniej cyfr w części całkowitej zostaną wyświetlone z użyciem zer początkowych.
`,#.##;'(',#.##')'`	W przypadku zera i wartości dodatnich wyświetla do dwóch miejsc po przecinku. Liczby ujemne należy umieścić w nawiasach okrągłych.
`'$'#,.00`	Wyświetla liczby poprzedzone znakiem `'$'` z dokładnością do dwóch miejsc po przecinku. Zawiera również separatory części tysięcznej.
`#.00;'('#.00')';'zero'`	Wyświetla zarówno dodatnie, jak i ujemne liczby z dokładnością do dwóch miejsc po przecinku. Liczby ujemne będą wyświetlane w nawiasach okrągłych. Dla wartości `0` wyświetlany jest łańcuch `zero`.

Poniższe kroki służą do zdefiniowania prostego formatu wyświetlania pola *TOTAL_VALUE* dla projektu, który został utworzony na początku rozdziału.

1. W oknie *Object Inspector* wybierz pole *TOTAL_VALUE*. W tym celu należy użyć listy obiektów okna *Object Inspector* i wybrać *SQLClientDataSet1TOTAL_VALUE* lub wybrać w edytorze pól pole stałe *TOTAL_VALUE*.
2. Ustaw dla właściwości *DisplayFormat* następującą wartość:

   ```
   ,#.00
   ```

3. Uruchom projekt, a następnie za pomocą poziomego paska przewijania przejdź do pola *TOTAL_VALUE*. Format wyświetlania tego pola widoczny na rysunku 8.6 powinien zawierać separator części tysięcznej oraz dwa miejsca po przecinku.

Form1

DATE_NEEDED	PAID	QTY_ORDERED	TOTAL_VALUE	DISCOUNT
08/17/1993	y	1000	560,000.00	0.20000000
09/01/1993	y	1	.00	
	y	2	600.50	
05/02/1993	n	5	20,000.00	
01/01/1994	n	150	14,850.00	0.05000000
10/01/1993	y	20	12,582.12	0.10000000
12/01/1993	y	900	27,000.00	0.30000001
01/24/1994	n	3	.00	
09/25/1993	y	1	100.02	
04/17/1993	y	1	47.50	
	y	40	399,960.50	0.10000000
11/11/1993	n	1	490.69	
	y	15	450,000.49	
03/01/1994	n	1	999.98	

Rysunek 8.6. *Właściwość DisplayFormat stosowana jest do kontroli wyświetlania zawartości pola numerycznego*

Właściwość *EditFormat* również wykorzystuje specyfikatory właściwości *DisplayFormat*. Podstawową różnicą jest to, że właściwość *EditFormat* ma wpływ na wyświetlanie pola tylko wtedy, gdy znajduje się ono w trybie edycji. Jeśli na przykład operacje wykonane w powyższych krokach, mające na celu zdefiniowanie formatu wyświetlania, zostałyby zastosowane wobec właściwości *EditFormat*, a nie *DisplayFormat*, wtedy aby zobaczyć wymagane dwa miejsca po przecinku, należałoby najpierw wejść w tryb edycji pola.

Tworzenie maski edycji

Maska edycji działa podobnie jak właściwości *DisplayFormat* i *EditFormat*, tym samym wpływając na sposób wyświetlania zawartości pola. Istnieje jednak kilka istotnych różnic. Po pierwsze, podczas gdy obydwie właściwości mają zastosowanie w przypadku pól numerycznych, właściwość *EditMask* dotyczy pól, które z reguły zawierają łańcuchy czyli pola tekstowe, pola daty, pola *memo*. Po drugie, obydwie właściwości kontrolują również sposób, w jaki wyświetlana jest zawartość pola, ale właściwość *EditMask* ma większy wpływ na to, co wprowadza użytkownik. Maska edycji może na przykład ograniczyć użytkownikowi możliwość wpisywania tylko liczb lub liter albo dokonywać zamiany wprowadzanych znaków na duże litery. Inaczej mówiąc, maska edycji może być stosowana do sprawdzania poprawności danych wprowadzanych przez użytkownika.

Ponieważ maska edycji realizuje znacznie więcej operacji niż zwykła maska wyświetlania, zatem specyfikatory stosowane w jej przypadku są bardziej złożone. Zostały one pokazane w tabeli 8.3.

W poprzedniej części powiedzieliśmy, że specyfikator formatu wyświetlania może zawierać jedną, dwie lub trzy części. Specyfikatory maski edycji natomiast zawsze posiadają trzy części, z których każda jest oddzielona średnikiem. Pierwsza część identyfikuje samą maskę. Druga decyduje o tym, czy domyślne znaki wstawiane przez maskę zapisywane są w powiązanym polu. Będzie ona zawierać 0 lub 1, gdzie 0 oznacza, że domyślne znaki nie zostaną zapisane w powiązanym polu, a 1, że zostaną. Wartości te mają również wpływ na właściwość *Text* pola. Właściwość *EditText* pola zawiera zawsze stałe łańcuchowe niezależnie od tego, jaka wartość została wstawiona w drugiej części specyfikatora.

Trzecia część specyfikatora maski edycji pozwala określić, jakie znaki drukowane zostaną wyświetlone jako reprezentacja wymaganych lub opcjonalnych znaków, które nie zostały jeszcze wprowadzone. Weźmy dla przykładu poniższą maskę edycji:

```
!\1 (000) 000-0000;1;_
```

Pozwala ona na wpisanie typowego numeru telefonu stosowanego w USA, gdzie za cyfrą `1` znajduje się spacja i nawias okrągły. Następnie użytkownik musi podać trzycyfrowy numer kierunkowy, po którym automatycznie wstawiany jest nawias okrągły. Dalej należy wpisać siedem cyfr numeru telefonu, gdzie znak '-' jest wstawiany automatycznie przez maskę edycji po trzeciej cyfrze. Cyfra `1` w drugiej części oznacza, że domyślne znaki zostaną zapisane w powiązanym polu, natomiast znak '_' określa, jaki drukowany znak zostanie wyświetlony w celu reprezentacji znaków jeszcze niewpisanych w polu posiadającym maskę edycji.

Tabela 8.3. *Specyfikatory stosowane do tworzenia maski edycji*

Specyfikator	Opis
`A`	W tym miejscu należy wstawić literę.
`a`	W tym miejscu można wstawić literę, ale nie jest to wymagane.
`C`	W tym miejscu należy wstawić dowolny znak drukowany.
`c`	W tym miejscu można wstawić drukowany znak, ale nie jest to wymagane.
`0`	W tym miejscu należy wstawić liczbę.
`9`	W tym miejscu można wstawić liczbę, ale nie jest to wymagane.
`#`	W tym miejscu można wstawić liczbę lub znaki '+', '-', ale nie jest to wymagane.
`L`	W tym miejscu należy wstawić literę lub liczbę.
`l`	W tym miejscu można wstawić literę lub liczbę, ale nie jest to wymagane.
`>`	Wszystkie litery wstawione po znaku '>' maski zostaną zamienione na duże litery.
`<`	Wszystkie litery wstawione po znaku '<' maski zostaną zamienione na małe litery.
`<>`	Dla wszystkich liter wstawionych po znakach '<>' maski nie zostanie dokonana zmiana ich wielkości.
`:`	Jest to separator czasu w formacie godzina:minuta:sekunda. O tym, jaki znak zostanie wyświetlony, decyduje jego ustawienie w systemie operacyjnym.
`/`	Jest to separator daty w formacie rok/miesiąc/dzień. O tym, jaki znak zostanie wyświetlony, decyduje jego ustawienie w systemie operacyjnym.
`\`	Dowolny znak, który zostanie wstawiony za tym znakiem, jest stałą łańcuchową i jako taki zostanie wyświetlony. Można go zastosować w celu dodania do maski dowolnego znaku, w tym specyfikatorów lub znaku średnika. Każdy znak, który nie jest wykorzystywany przez maskę, może zostać w niej wstawiony bez używania znaku '\' i będzie traktowany jak stała łańcuchowa.
`!`	Wszystkie znaki wstawione w miejsce tego znaku we właściwości *DisplayText* pola zostaną wyświetlone w postaci pustych miejsc umieszczonych na początku. W przypadku opuszczenia tego znaku we właściwości *DisplayText* pola zostaną wyświetlone puste miejsca umieszczone na końcu. Podczas edycji pola puste miejsca zastępowane są przez drukowane znaki zdefiniowane w trzeciej części specyfikatora.

Jeśli przyjrzymy się zawartości pola *PO_NUMBER* tabeli *SALES*, zauważymy, że zawsze zaczyna się znakiem '`V`', po którym występują kolejno: dwie liczby, liczba lub litera i na końcu cztery wymagane litery. Poniższy specyfikator maski edycji określa taki format (znaki '`0`' są zerami):

```
!>V00C0000;0;*
```

Kolejne kroki dodają powyższą maskę edycji do pola stałego *PO_NUMBER*. Zastosowany projekt został utworzony na początku tego rozdziału.

1. W oknie *Object Inspector* wybierz pole *TOTAL_VALUE*. W tym celu należy użyć listy obiektów okna *Object Inspector* i wybrać *SQLClientDataSet1PO_NUMBER* lub wybrać w edytorze pól pole stałe *PO_NUMBER*.
2. Ustaw dla właściwości *EditMask* poniższą wartość:

   ```
   !>V00C0000;0;*
   ```

3. Uruchom projekt. Aby wstawić nowy rekord, kliknij w komponencie *DBNavigator* przycisk +. W polu *PO_NUMBER* wejdź w tryb edycji dla dodanego rekordu, wyświetlonego przez kontrolkę *DBGrid*. Na rysunku 8.7 widoczna jest zdefiniowana maska edycji.

Rysunek 8.7. *Po wejściu w tryb edycji pola PO_NUMBER pojawia się powiązana z nim maska edycji*

Form1

	PO_NUMBER	CUST_NO	SALES_REP	ORDER_STATUS	ORDER_DATE	SHIP_DATE
*	V*******					
	VV91E0210	1004	11	shipped	03/04/1991	03/05/1991
	VV92E0340	1004	11	shipped	10/15/1992	10/16/1992
	VV92F3004	1012	11	shipped	10/15/1992	01/16/1993
	VV92J1003	1010	61	shipped	07/26/1992	08/04/1992
	VV9320630	1001	127	open	12/12/1993	
	VV9324200	1001	72	shipped	08/09/1993	08/09/1993
	VV9324320	1001	127	shipped	08/16/1993	08/16/1993
	VV9333005	1002	11	shipped	02/03/1993	03/03/1993
	VV9333006	1002	11	shipped	04/27/1993	05/02/1993
	VV9336100	1002	11	waiting	12/27/1993	01/01/1994
	VV9345139	1003	127	shipped	09/09/1993	09/20/1993
	VV9345200	1003	11	shipped	11/11/1993	12/02/1993
	VV9346200	1003	11	waiting	12/31/1993	

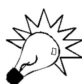
4. Aby wyjść z trybu edycji lub anulować operację wstawiania rekordu, a następnie zamknąć projekt, należy dwa razy nacisnąć klawisz *Esc*.

Kontrola dostępu do pola

Mimo że dla zestawu danych zostały utworzone pola stałe, możemy spotkać się z koniecznością większej kontroli dostępu do tych pól. W tym celu zastosujemy dwie właściwości pól: *Visible* oraz *ReadOnly*.

Po ustawieniu wartości właściwości *Visible* na *False* pole nie zostanie wyświetlone w kontrolkach bazodanowych. Jeśli nie stosujemy kolumn stałych w komponencie *DBGrid*, wtedy wszystkie pola zestawu danych wyświetlane są w siatce. W takiej sytuacji można ustawić wartość właściwości *Visible* na *False*, co spowoduje, że pole nie zostanie wyświetlone w komponencie *DBGrid*.

Może również zaistnieć konieczność zablokowania użytkownikowi możliwości zmiany wartości niektórych kolumn przy jednoczesnym ich wyświetleniu. W tym celu można posłużyć się właściwością *ReadOnly* pola. Gdy wartość tej właściwości jest ustawiona na *True*, wtedy użytkownik może przeglądać wartości pola, ale nie ma możliwości ich zmiany.



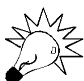
Jeśli określone pole tabeli jest wymagane, ale jego właściwość *ReadOnly* ma wartość *True*, wtedy użytkownik będzie w stanie wstawić nowy rekord, ale może nie móc go zatwierdzić, ponieważ nie będzie mógł wstawić wartości w tym polu.

Walidacja pól przy użyciu ograniczeń pola

Ograniczenia dostarczają mechanizmów prostego definiowania w fazie projektowania reguł walidacji pola, które muszą być spełnione przez użytkownika. Walidacja ta związana jest z zasadami, które stosowane są przez aplikację Kyliksa w stosunku do pola, jeszcze zanim jego wartość zostanie zapisana w powiązanym buforze rekordu. Jeśli wartość pola narusza zasadę, wtedy wywoływany jest wyjątek i wartość ta nie może zostać zapisana w buforze rekordu.

W Kyliksie w fazie projektowania można określić dwa typy ograniczeń: na poziomie pola oraz na poziomie zestawu danych. Ograniczenia na poziomie zestawu danych, omówione w następnym rozdziale, dostarczają mechanizmów definiowania walidacji na poziomie rekordu (walidacja na poziomie rekordu realizowana jest podczas zatwierdzania całego rekordu). W tej części omówimy ograniczenia na poziomie pola.

Ograniczenia na poziomie pola określane są przy użyciu wyrażeń logicznych SQL, które wykonywane są, zanim wartość zostanie wstawiona do pola. Jeśli wyrażenie będzie miało wartość *False*, wtedy zostanie wywołany wyjątek, który nie pozwoli na zatwierdzenie zmian w polu.



Zdarzenie *OnValidate* pola dostarcza innego mechanizmu definiowania walidacji na poziomie pola. Obsługa zdarzenia *OnValidate* zostanie opisana w dalszej części rozdziału.

Przy definiowaniu ograniczeń na poziomie pola należy również skonfigurować dwie właściwości pola: *CustomConstraint* oraz *ConstraintErrorMessage*. Właściwości *CustomConstraint* przypisywane jest wyrażenie logiczne SQL, natomiast właściwość *ConstraintErrorMessage* określa tekst wyjątku, który zostanie wywołany, jeśli ograniczenie nie zostanie spełnione.

Wyrażenie SQL typu boole'owskiego dotyczy tylko pola, do którego zostało przypisane ograniczenie. Ściśle mówiąc, wyrażenia takie mogą się tylko odnosić do wartości wstawionej w polu rekordu znajdującego się w buforze, w funkcjach SQL lub operatorach porównania i stałych. Wyrażenie SQL nie może odnosić się do wartości innych pól tego samego rekordu. Aby utworzyć ograniczenie, które dotyczy wartości dwóch lub więcej pól tego samego rekordu, należy zastosować ograniczenia na poziomie zestawu danych lub posłużyć się procedurą obsługi zdarzenia *OnValidate*.

Nazwa, która zostanie użyta w wyrażeniu typu boole'owskiego do odwoływania się do pola jest dowolna, jednak nie może być jednym z zastrzeżonych słów kluczowych języka SQL. Na przykład poniższe dwa łańcuchy po wstawieniu do właściwości *CustomConstraint* pola spowodują identyczny efekt.

```
PO_NUMBER IS NOT NULL
x IS NOT NULL
```

Nie ma tu znaczenia, że `x` nie jest nazwą pola powiązanego zestawu wynikowego, ponieważ Kylix automatycznie stwierdza, że ma być wykonane odwołanie do wartości pola, z którym powiązane jest ograniczenie. Co więcej, łańcuch *PO_NUMBER* zachowa się identycznie, co oznacza, że wyrażenie to jest poprawne, nawet jeśli pole, do którego jest ono przypisane, nie ma nazwy *PO_NUMBER*.

Poniższe kroki pokazują, jak dodać ograniczenia do projektu, który został utworzony na początku rozdziału.

1. W oknie *Object Inspector* wybierz pole *DISCOUNT*. W tym celu należy użyć listy obiektów okna *Object Inspector* i wybrać *SQLClientDataSet1DISCOUNT* lub wybrać w edytorze pól pole stałe *DISCOUNT*.
2. Ustaw dla właściwości *CustomConstraint* poniższe wyrażenie typu boole'owskiego:

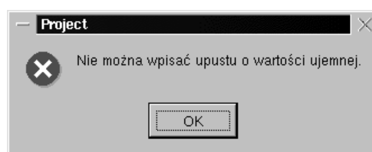
   ```
   dis >= 0
   ```

3. Następnie dodaj do właściwości *ConstraintErrorMessage* poniższy komunikat:

   ```
   Nie można wpisać upustu o wartości ujemnej.
   ```

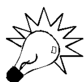
4. Uruchom aplikację. Wstaw nowy rekord, a następnie przejdź do pola *DISCOUNT* i wpisz `-.1`. Aby przejść do następnego pola, naciśnij klawisz *Tab*. Kylix wyświetli komunikat z następującym wyjątkiem (patrz rysunek 8.8):

Rysunek 8.8.
Wyjątek



5. Zamknij aplikację.

Oprócz właściwości *CustomConstraint* większość pól posiada właściwość *Required*. Gdy ma ona wartość *True*, wtedy użytkownik nie może zatwierdzić aktualnego rekordu, jeśli w wymaganym polu nie została wstawiona wartość. Podobnie w przypadku pól numerycznych można właściwościom *Max* i *Min* przypisać wartości w celu określenia maksymalnej i minimalnej dopuszczalnej wartości pola.



Wartości określone dla właściwości *CustomConstraint*, *Min*, *Max* oraz *Required* pola dotyczą ustawień po stronie klienta. Więzy integralnościowe w systemie RDBMS również mogą definiować, jakie wartości poszczególnych pól są dopuszczalne. W przypadku zaistnienia konfliktu ustawień właściwości pola i więzów integralności zdefiniowanych po stronie serwera, użytkownik nie będzie mógł dodać lub zaktualizować danych powiązanego zestawu danych.

Praca z polami

Wszystkie operacje wykonane w fazie projektowania w środowisku IDE mogą być zrealizowane również w fazie wykonania. Można na przykład w fazie projektowania w Kyliksie ustawić publikowane właściwości, takie jak *CustomConstraint*, *DisplayFormat*, lub po uruchomieniu w postaci kodzie metody lub procedury. Jednak odwrotna zależność nie zawsze jest możliwa. Ściśle mówiąc, pewnych operacji, które można wykonać po uruchomieniu nie da się zrealizować w fazie projektowania. Należą do nich: wstawianie w polu nowych wartości, zapisywanie wartości pola na dysku (pola *memo* lub *blob*) oraz określenie rozmiaru danych pola.

W dalszej części rozdziału zostaną omówione operacje na polach, które można wykonać w fazie wykonania. Należą do nich: odczyt i zapis danych pól powiązanej bazy danych oraz walidacja wartości pola przy użyciu procedury obsługi zdarzenia *OnValidate*.

Odczyt i zapis pól zestawu danych

Operacje odczytu i zapisu danych pól zestawu danych realizowane są przy użyciu właściwości pola. Najwygodniejsza jest właściwość `Value` typu `Variant`. Inne dostępne właściwości to: *AsString*, *AsInteger*, *AsFloat* oraz *AsBoolean*. Były one początkowo stosowane w Delphi, zanim nie została wprowadzona obsługa typu `Variant`. W Kyliksie zaleca się stosować jak najczęściej właściwość *Value*. Jeśli jednak z jakiegoś powodu użycie tej właściwości stwarza problemy, można wtedy sięgnąć po jedną z właściwości *As...*.

Aby odczytać wartość pola, zestaw danych, do którego ono należy, musi być aktywny oraz musi zwracać zestaw danych (polecenia DDL, takie jak `create table`, nie zwracają danych ani mają pól). Poniższy kod prezentuje odczyt wartości pola, który przypisuje zmiennej `OrdStat` typu `String` wartość pola *ORDER_STATUS*.

```
var
  OrdStat: String;
begin
  OrdStat := SQLClientDataSet1.FieldbyName('ORDER_STATUS').Value;
```

Aby dokonać zapisu pola, zestaw danych musi znajdować się w jednym z dwóch stanów: *dsEdit* lub *dsInsert*. Jeśli jest to pole wyliczeniowe, wtedy zapis do niego możliwy jest tylko w stanach *dsCalcFields* lub *dsInternalCalc*. Do określenia aktualnego stanu zestawu danych służy właściwość *State*, natomiast metody `Edit` lub `Insert` służą do zmiany jego stanu, jeśli jest to możliwe. Zestaw danych, którego właściwość *CanModify* zwraca wartość *False*, nie może być modyfikowany. Uruchomienie metody `Edit` lub `Insert` na takim zestawie danych spowoduje wywołanie wyjątku.

Zakładając, że pole *ORDER_STATUS* jest piątym polem powiązanego zestawu danych, poniższy kod przypisuje wartość `shipped` temu polu dla aktualnego rekordu.

```
SQLClientDataSet1.Fields[4].value := 'shipped';
```

Indeks właściwości *Fields* liczony jest od zera. Aby odwołać się do pierwszego pola, należy użyć zera, a w przypadku piątego (jak w tym przykładzie) indeks ma wartość 4.

Jeśli utworzono pola stałe, można się nimi posłużyć do bezpośredniego odwoływania się do pól zamiast stosowania odwoływania za pomocą zestawu danych.

W projekcie modyfikowanym w kilku przykładach tego rozdziału utworzono jedno pole stałe dla każdego pola powiązanego zestawu danych. Odwołanie przez pole stałe do pola *ORDER_STATUS* ma nazwę *SQLClientDataSet1ORDER_STATUS*. Poprzednią linię kodu można zastąpić poniższą:

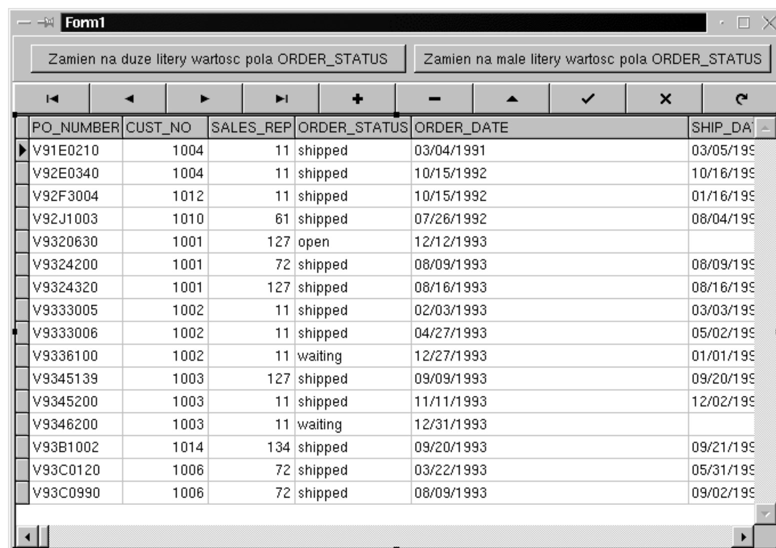
```
SQLClientDataSet1ORDER_STATUS.Value := 'shipped';
```

Jeśli przyjrzymy się obu liniom kodu, zauważymy, że ta, która odwołuje się bezpośrednio przez pole stałe zawiera więcej znaków, co wiąże się z większą pracochłonnością. Jest tak tylko dlatego, że do pola stałego została przypisana domyślna nazwa. Po dodaniu pola stałego do edytora pól można w oknie *Object Inspector* zmienić wartość właściwości przechowującej jego nazwę. Po wybraniu krótkiej nazwy, takiej jak *OS_FLD*, kod również będzie znacznie krótszy, co widać w poniższym przykładzie.

```
OS_FLD.Value := 'shipped';
```

Poniższy przykład prezentuje, w jaki sposób po uruchomieniu odczytywać i zapisywać pola. Zawiera on typową operację polegającą na przeszukiwaniu zestawu danych i pracy z jego wybranymi rekordami. W przykładzie zostaną utworzone dwa przyciski, każdy z przypisaną procedurą obsługi zdarzenia *OnClick*. Jeśli pierwszy z nich zostanie kliknięty, wtedy wartości pola *ORDER_STATUS* zostaną zamienione na duże litery, natomiast jeśli klikniemy drugi przycisk, wtedy wartości zostaną zamienione na małe litery.

1. W projekcie, który był już używany w tym rozdziale, dodaj dwa przyciski w górnym panelu.
2. Ustaw wartość właściwości *Caption* przycisku *Button1* na *Zamień na duże litery wartość pola ORDER_STATUS*, a przycisku *Button2* na *Zamień na małe litery wartość pola ORDER_STATUS*. Formularz powinien teraz wyglądać podobnie jak na rysunku 8.9.



PO_NUMBER	CUST_NO	SALES_REP	ORDER_STATUS	ORDER_DATE	SHIP_DA
V91E0210	1004	11	shipped	03/04/1991	03/05/199
V92E0340	1004	11	shipped	10/15/1992	10/16/199
V92F3004	1012	11	shipped	10/15/1992	01/16/199
V92J1003	1010	61	shipped	07/26/1992	08/04/199
V9320630	1001	127	open	12/12/1993	
V9324200	1001	72	shipped	08/09/1993	08/09/199
V9324320	1001	127	shipped	08/16/1993	08/16/199
V9333005	1002	11	shipped	02/03/1993	03/03/199
V9333006	1002	11	shipped	04/27/1993	05/02/199
V9336100	1002	11	waiting	12/27/1993	01/01/199
V9345139	1003	127	shipped	09/09/1993	09/20/199
V9345200	1003	11	shipped	11/11/1993	12/02/199
V9346200	1003	11	waiting	12/31/1993	
V93B1002	1014	134	shipped	09/20/1993	09/21/199
V93C0120	1006	72	shipped	03/22/1993	05/31/199
V93C0990	1006	72	shipped	08/09/1993	09/02/199

Rysunek 8.9. *Dwa przyciski dodane do projektu używanego w przykładach zawartych w tym rozdziale*

3. Dodaj procedurę obsługi zdarzenia *OnClick* do przycisku *Button1*. Wprowadź do niej poniższy kod:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  with DataModule2.SQLClientDataSet1 do
  begin
    DisableControls;
    try
```

```
      First;
      while not EOF do
      begin
        try
          Edit; //aktywacja stanu dsEdit
          FieldByName('ORDER_STATUS').Value :=
            UpperCase(FieldByName('ORDER_STATUS').Value);
          Post;
        except
          //Rekord jest używany. Ignoruj.
        end; // try
        Next;
      end; // while not eof
    finally
      EnableControls;
    end; //try
  end; //with SQLClientDataSet1 do
end;
```

4. Następnie dodaj procedurę obsługi zdarzenia *OnClick* do przycisku *Button2* i wprowadź do niej poniższy kod. W celu przyspieszenia operacji można skopiować zawartość poprzedniej procedury i zmienić wywoływanie metody `Uppercase` na `Lowercase`:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  with DataModule2.SQLClientDataSet1 do
  begin
    DisableControls;
    try
      First;
      while not EOF do
      begin
        try
          Edit; // aktywacja stanu dsEdit
          FieldByName('ORDER_STATUS').Value :=
            LowerCase(FieldByName('ORDER_STATUS').Value);
          Post;
        except
          // Rekord jest używany. Ignoruj.
        end; // try
        Next;
      end; // while not eof
    finally
      EnableControls;
    end; //try
  end; //with SQLClientDataSet1 do
end;
```

5. Zapisz zmiany dokonane w projekcie i następnie uruchom go.

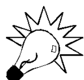
Aby w uruchomionej aplikacji zamienić tekst w polu *ORDER_STATUS* na duże litery, należy kliknąć przycisk o nazwie *Zamień na duże litery wartość pola ORDER_STATUS*, tak jak to pokazano na rysunku 8.10. Aby natomiast przywrócić zawartość tego pola do stanu początkowego, należy kliknąć przycisk *Zamień na małe litery wartość pola ORDER_STATUS*.

Rysunek 8.10.
Kod powiązany z przyciskiem Zamień na duże litery wartość pola ORDER_STATUS przeszukuje rekordy zestawu danych i zamienia zawartość pola ORDER_STATUS na duże litery

Form1

Zamien na duze litery wartosc pola ORDER_STATUS | Zamien na male litery wartosc pola ORDER_STATUS

PO_NUMBER	CUST_NO	SALES_REP	ORDER_STATUS	ORDER_DATE	SHIP_DATE
V93F0020	1009	61	SHIPPED	10/10/1993	11/11/1993
V93F2030	1012	134	OPEN	12/12/1993	
V93F2051	1012	134	WAITING	12/18/1993	
V93F3088	1012	134	SHIPPED	08/27/1993	09/08/1993
V93H0030	1005	118	OPEN	12/12/1993	
V93H0500	1008	61	OPEN	12/12/1993	
V93H3009	1008	61	SHIPPED	08/01/1993	12/02/1993
V93I4700	1013	121	OPEN	10/27/1993	
V93J2004	1010	118	SHIPPED	10/30/1993	12/02/1993
V93J3100	1010	118	SHIPPED	08/20/1993	08/20/1993
V93N5822	1015	134	SHIPPED	12/18/1993	01/14/1994
V93S4702	1011	121	SHIPPED	10/27/1993	10/28/1993
V9420099	1001	127	OPEN	01/17/1994	
V9427029	1001	127	SHIPPED	02/07/1994	02/10/1994
V9456220	1007	127	OPEN	01/04/1994	
V94H0079	1005	61	OPEN	02/13/1994	
V94S6400	1011	141	WAITING	01/06/1994	



Wywołanie w tym przykładzie metody `DisableControls` w przypadku obydwu procedur obsługi zdarzeń tymczasowo wyłączy komunikację między zestawem danych i jego źródłem danych. Pozwala to na szybkie przeszukanie zestawu danych bez konieczności aktualizowania zawartości każdej kontrolki bazodanowej, która wskazuje na źródło danych powiązane z tym zestawem. W przypadku stosowania metody `DisableControls` należy dołączyć słowa kluczowe `try-finally` oraz wywołać metodę `EnableControls`. W ten sposób mamy pewność, że komunikacja między zestawem danych i źródłem danych zostanie przywrócona, nawet jeśli po wywołaniu metody `DisableControls` wystąpi wyjątek.

Obsługa zdarzenia OnValidate

Na początku rozdziału omówiono tworzenie własnych ograniczeń, które realizują walidację na poziomie pola. W przypadku takich ograniczeń można stosować tylko jedno wyrażenie logiczne SQL oraz jeden komunikat błędu. Zdarzenie *OnValidate* egzemplarza pola jest drugim sposobem na zastosowanie walidacji na poziomie pola. Za jego pomocą można wykonywać wielokrotne testy działania walidacji oraz zdefiniować więcej niż jeden komunikat błędu, w zależności od tego, jaki problem związany z walidacją wystąpił.

Procedura obsługi zdarzenia *OnValidate* pola wykonywana jest zanim nowa wartość pola zostanie wprowadzona do powiązanego bufora rekordu. Następuje to w momencie, gdy użytkownik przechodzi z kontrolki bazodanowej powiązanej z tym polem do innej, zamyka komórkę komponentu *DBGrid* połączoną z polem lub zatwierdza cały rekord.

Przy użyciu procedury obsługi zdarzenia można wykonać dowolne testy, włączając w to: odczyt wartości innych pól tego samego rekordu, użycie innego zestawu danych do odwoływania się do wartości innego zestawu wynikowego, sprawdzanie wartości kontrolek interfejsu użytkownika, które nie są bazodanowe. Jeśli kod stwierdzi, że wartość pola jest z jakiegoś powodu niedopuszczalna, można wywołać wyjątek z komunikatem błędu wyjaśniającym dlaczego wartość jest nieprawidłowa.

Poniższe kroki opisują operację definiowania walidacji na poziomie pola przy użyciu procedury obsługi zdarzenia *OnValidate*. W tym przypadku procedura ta użyta dla pola *DATE_NEEDED* sprawdzi, czy użytkownik wprowadził poprawną datę, czyli datę która nie jest datą z przeszłości.

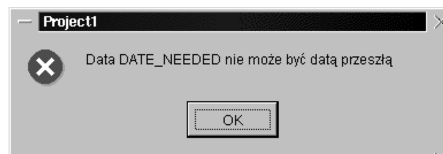
1. Otwórz moduł danych projektu, który był wykorzystywany w poprzednich przykładach rozdziału.
2. Dodaj poniższą definicję typu w części `interface` modułu. Zostały w niej zdeklarowane dwie nowe klasy `Exception`. Pierwsza z nich — `ECustomException` — jest klasą nadrzędną lub superklasą wszystkich zdefiniowanych wyjątków, które będzie można wywołać w kodzie. W rozdziale 5. wspomniano, że takie podejście ułatwia zarządzanie wyjątkami.

```
type
 ECustomException = class(Exception);
 EInvalidFieldException = class(ECustomException);
```

3. W edytorze pól wybierz pole *DATE_NEEDED*.
4. Wyświetl zakładkę *Events* okna *Object Inspector* i utwórz procedurę obsługi zdarzenia *OnValidate*.
5. Wprowadź do niej poniższy kod:

```
procedure TDataModule2.SQLClientDataSet1DATE_NEEDEDValidate(
 Sender: TField);
begin
if Sender.AsDateTime < Date then
 raise EInvalidFieldException.Create('Data DATE_NEEDED nie może być '+
  'datą przeszłą.');
end;
```

6. Zapisz projekt i uruchom go. Wstaw nowy rekord i podaj datę starszą niż data aktualna, którą wyświetla zegar systemowy. W celu przejścia do następnego rekordu naciśnij klawisz prawej strzałki. Zostanie wtedy wywołany następujący wyjątek (patrz rysunek 8.11).



Rysunek 8.11. *Wyjątek*

7. Zamknij okno dialogowe i spróbuj przejść do innego rekordu. Spowoduje to próbę zatwierdzenia rekordu, co z kolei rozpocznie operację walidacji pola i ostatecznie wywoła wyjątek. Jeśli nie podasz poprawnej daty, wtedy nie będziesz mógł opuścić pola lub zatwierdzić rekordu.

Dostęp do pola — wydajność i zarządzanie

Jak napisaliśmy w poprzedniej części, można wyróżnić trzy sposoby dostępu do pól stałych — za pomocą właściwości *Fields* zestawu danych, metody `FieldByName` lub odwołania do pola stałego. Czy ma jakiekolwiek znaczenie to, którego sposobu użyjemy?

Odpowiedź brzmi — tak. Właściwie jest to bardziej skomplikowane, niż mogłoby się wydawać. Ściśle mówiąc, każdy sposób ma swoje wady i zalety. W rezultacie aby wybrać najbardziej odpowiednią technikę dostępu do pól stałych, należy przeanalizować wymagania aplikacji.

Wady i zalety powyższych technik omówimy w dalszej części rozdziału. Na końcu zaprezentujemy, jak można połączyć dwa z powyższych sposobów w celu osiągnięcia maksymalnej wydajności i łatwości obsługi kodu.

Zastosowanie właściwości Fields

Właściwość *Fields* pozwala na bezpośredni dostęp do powiązanego pola. W przeciwieństwie do metody `FieldByName` właściwość ta nie powoduje dodatkowego obciążenia związanego z przeszukiwaniem powiązanego pola. W efekcie wydajność kodu wykorzystywanego przez właściwość *Fields* jest lepsza.

Niestety kod ten jest trudny do zanalizowania, ponieważ podczas dostępu do określonego pola dokonywane jest odwołanie do niego za pomocą liczby. Ponadto zmiany w strukturze powiązanej tabeli lub kolejności pól w poleceniu `select` może spowodować odwołanie do niewłaściwego pola. Podsumowując, kod wykorzystywany przez właściwość *Fields* jest trudniejszy w konserwacji i debugowaniu.

Zastosowanie metody FieldByName

Metoda `FieldByName` działa wolniej niż właściwość *Fields*, ponieważ w celu odnalezienia numeru porządkowego pola podanego w tej właściwości musi ona przeszukać listę pól zestawu danych. Wzrost obciążenia w tym przypadku jest wprost proporcjonalny do częstotliwości wywoływania metody `FieldByName`. Jednorazowe uruchomienie metody nie stwarza problemu, ale przy ilości 100 000 razy — już tak.

Mimo że metoda `FieldByName` nie prezentuje się najlepiej pod względem wydajnościowym, to jednak kod przez nią wykonywany jest łatwy do analizy, ponieważ nazwa pola, do którego jest realizowany dostęp, pojawia się przy jej wywołaniu. W efekcie często łatwiej jest zarządzać kodem, który wykonuje metoda `FieldByName`, i debugować go.

Ponadto w przypadku tej metody zmiana kolejności pól powiązanego zestawu danych nie ma znaczenia. Tak długo jak nazwy pól się nie zmienią, zmiana ich kolejności w tabeli lub w poleceniu `select`, nie będzie wymagała dokonania żadnych zmian w kodzie wykorzystującym metodę `FieldByName`.

Z drugiej strony metoda ta jest bardzo wrażliwa na zmiany nazw pól. Za każdym razem, gdy zmienia się nazwa pola, do którego metoda `FieldByName` ma dostęp, należy odpowiednio zmodyfikować jej kod. Jednak błędy, które pojawiają się w związku ze zmianą nazwy pola, są stosunkowo łatwe do wykrycia. Nazwa pola, które nie może być znalezione, pojawia się w komunikacie wyjątku.

Zastosowanie odwołań do pól stałych

Odwołania do pól stałych oferują zalety zarówno właściwości *Fields*, jak i metody `FieldByName`. Podobnie jak właściwość *Fields*, odwołania do pól stałych są odwołaniami bezpośrednimi do pola, a zatem nie wymagają przeszukiwania, jak ma to miejsce

w przypadku metody `FieldByName`. Ponadto ponieważ nazwy pól stałych oparte są na nazwach powiązanych pól, kod wykorzystujący odwołania do pól stałych jest zwykle czytelniejszy i łatwiejszy do zanalizowania.

Jednak w przeciwieństwie do właściwości *Fields* i metody `FieldByName`, które mogą być używane zarówno przez domyślne pola, jak i pola stałe, w przypadku odwołań do pól stałych wymagane jest ich określenie. Jak wspomniano wcześniej, zmiana w strukturze powiązanej tabeli lub kolejności pól w poleceniu `select` czasem wiąże się z usunięciem i ponownym określeniem pól stałych, włączając w to ponowną konfigurację właściwości oraz procedur obsługi zdarzeń. W przypadku małej bazy danych nie jest to kłopotliwe, ale przy znacznej ilości tabel może być nie do zaakceptowania.

Zastosowanie zmiennych pola

Można wyróżnić jeszcze jedną technikę dostępu do pól, która do tej pory nie została omówiona. Jest ona efektem połączenia dwóch z trzech technik wymienionych wcześniej. Zawiera dwa składniki. Pierwszy z nich to deklaracja jednej zmiennej dla każdego pola, do którego aplikacja będzie żądała dostępu. Drugi wiąże się z przypisaniem tej zmiennej odwołania do pola zwróconego przez właściwość *Fields* lub metodę `FieldByName`.

Mimo, że taki sposób dostępu do pola wymaga na początku napisania znacznej ilości kodu, to w zamian później uzyskamy dwie istotne korzyści. Pierwsza z nich to umieszczenie wszystkich wartości przypisanych do zmiennych pól w jednym miejscu, takim jak procedura obsługi zdarzenia *OnCreate* modułu danych. Jeśli kolejne zmiany dokonywane są w powiązanej strukturze jednej lub kilku tabel, wtedy wystarczy tylko zmodyfikować przypisania zmiennych. Wszystkie pozostałe odwołania do pól pozostaną niezmienione.

Druga korzyść polega na tym, że do bezpośredniego odwoływania się do każdego pola można używać zrozumiałych nazw zmiennych. Ponieważ zmienne bezpośrednio odwołują się do powiązanych pól, wykonywany kod umożliwia równie szybki dostęp, jak ma to miejsce w przypadku właściwości *Fields* lub odwołań do pól stałych.

Przyjrzyjmy się komponentowi *SQLClientDataSet*, który znajduje się na module danych projektu stosowanego we wcześniejszych przykładach tego rozdziału. Ten moduł danych może zastosować technikę opisaną powyżej przez użycie dwóch segmentów kodu. Pierwszy z nich to deklaracja `var` dodawana do sekcji `implementation` modułu danych:

```
var
 DataModule2: TDataModule2;
 SALESPO_NUMBER: TStringField;
 SALESCUST_NO: TIntegerField;
 SALESSALES_REP: TSmallintField;
 SALESORDER_STATUS: TStringField;
 SALESORDER_DATE: TSQLTimeStampField;
 SALESSHIP_DATE: TSQLTimeStampField;
 SALESDATE_NEEDED: TSQLTimeStampField;
 SALESPAID: TStringField;
 SALESQTY_ORDERED: TIntegerField;
 SALESTOTAL_VALUE: TBCDField;
 SALESDISCOUNT: TFloatField;
 SALESITEM_TYPE: TStringField;
 SALESAGED: TFloatField;
```

Drugi segment zawiera poniższe przypisania zmiennych, które dodawane są do procedury obsługi zdarzenia *OnCreate* modułu danych:

```
procedure TDataModule2.DataModuleCreate(Sender: TObject);
begin
with SQLClientDataSet1 do
begin
 SALESPO_NUMBER :=
  FieldByName('PO_NUMBER') as TStringField;
 SALESCUST_NO :=
  FieldByName('CUST_NO') as TIntegerField;
 SALESSALES_REP :=
  FieldByName('SALES_REP') as TSmallintField;
 SALESORDER_STATUS :=
  FieldByName('ORDER_STATUS') as TStringField;
 SALESORDER_DATE :=
  FieldByName('ORDER_DATE') as TSQLTimeStampField;
 SALESSHIP_DATE :=
  FieldByName('SHIP_DATE') as TSQLTimeStampField;
 SALESDATE_NEEDED :=
  FieldByName('DATE_NEEDED') as TSQLTimeStampField;
 SALESPAID :=
  FieldByName('PAID') as TStringField;
 SALESQTY_ORDERED :=
  FieldByName('QTY_ORDERED') as TIntegerField;
 SALESTOTAL_VALUE :=
  FieldByName('TOTAL_VALUE') as TBCDField;
 SALESDISCOUNT :=
  FieldByName('DISCOUNT') as TFloatField;
 SALESITEM_TYPE :=
  FieldBYName('ITEM_TYPE') as TStringField;
 SALESAGED :=
  FieldByName('AGED') as TFloatField;
end;
end;
```

W celu uzyskania odwołań do pól procedura obsługi zdarzenia *OnCreate* mógłby również wykorzystać właściwość *Fields*. Jednak metoda `FieldByName` jest bardziej czytelna i łatwiejsza w obsłudze. Ponadto ponieważ takie przypisanie i przeszukiwanie powiązanego pola występuje tylko raz dla każdego pola, zatem ogólny wpływ metody `FieldByName` na spadek wydajności jest niezauważalny.

Odtąd wszystkie odwołania do pól zestawu danych mogą być wykonywane przy użyciu zmiennych pola. Na przykład segment kodu dokonujący konwersji na duże litery z poprzedniej części będzie wyglądać następująco:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
with DataModule2.SQLClientDataSet1 do
begin
 DisableControls;
 try
  First;
  while not EOF do
  begin
   try
    Edit; //aktywacja statusu dsEdit
    SALESORDER_STATUS.Value :=
     UpperCase(SALESORDER_STATUS.Value);
```

```
      Post;
    except
      //Rekord jest używany. Ignoruj.
    end; // try
    Next;
  end; // while not eof
 finally
  EnableControls;
 end; //try
end; //with SQLClientDataSet1 do
end;
```

Chociaż zastosowanie tej techniki dostępu do pola wymaga większej ilości kodu, dokonywanie dowolnych zmian w strukturze tabeli może być realizowane całkowicie w samym module danych, co eliminuje potrzebę modyfikowania odwołań do pól w innych elementach aplikacji. Godny uwagi jest fakt, że jeśli struktura danych często się zmienia, wtedy w celu ułatwienia zarządzania aplikacją można zastosować zmienne pola.

Tworzenie nowych pól stałych

Wszystkie pola, z którymi do tej pory mieliśmy do czynienia, były powiązane z polami z pewnego zestawem danych pobieranego z bazy danych. Nowe pola stałe są trochę inne. Chociaż mogą one być powiązane z danymi wynikowego zestawu, wartości, na które pola te wskazują nie są właściwie w nich przechowywane w ten sam sposób, w jaki ma to miejsce w przypadku innych typów pól. Zamiast tego, dane zawarte w nowych polach stałych są wynikiem wykonania operacji. Operacja ta może być wykonana przez same nowe pola stałe, tak jak ma to miejsce w przypadku pól wyszukiwania lub przy użyciu procedury obsługi zdarzenia, która jest stosowana dla pól obliczeniowych. Nowe pola stałe są tylko do odczytu, ponieważ ich dane nie są przechowywane w tabeli.

Zależnie od typu zestawu danych można wyróżnić pięć rodzajów nowych pól stałych:

- danych,
- obliczeniowe,
- wyszukiwania,
- wewnętrznych obliczeń,
- agregacji.

Komponenty *ClientDataSet* oraz *SQLClientDataSet* obsługują wszystkie wymienione typy. Inne klasy pochodne klasy `TDataSet`, które pojawią się w kolejnych wersjach Kyliksa lub w produktach innych producentów, mogą ich wszystkich nie obsługiwać.

Pole danych służy do definiowania pola stałego, którego typ danych, a przez to również instancja klasy `TField` różnią się od pola stałego tworzonego w normalnej sytuacji automatycznie. Na przykład wynikowy zestaw może zawierać pole o typie danych `LongInt`, ale jeśli chcemy, aby pole stałe było typu `ShortInt`, można wtedy zastosować pole danych.

Wartość pola obliczeniowego określana jest po uruchomieniu aplikacji w oparciu o procedurę obsługi zdarzenia *OnCalcFields*. Procedura obsługi zdarzenia *OnCalcFields* do określenia wartości pola wykorzystuje dane jednego lub kilku pól wynikowego zestawu. Na przykład pole obliczeniowe może zawierać połączenie wartości pól przechowujących imiona i nazwiska lub wyświetlać datę zapłaty faktury obliczoną na podstawie wartości pola tabeli zawierającego datę sprzedaży oraz pola określającego liczbę dni, które ma kupujący do zapłaty.

Pole wyszukiwania wyświetla dane z innego powiązanego wynikowego zestawu. Na przykład tabela *SALES* zawiera pole *SALES_REP*, przechowujące numer pracownika, który dokonał operacji sprzedaży. Ponieważ w bazie danych znajduje się również tabela *EMPLOYEE* zawierająca pole *EMP_NO*, którego wartości odpowiadają wartościom pola *SALES_REP* i posiada też pole *FULL_NAME*, można wtedy utworzyć pole wyszukiwania dla tabeli *SALES*, które wyświetli wartości pola *FULL_NAME*.

Pole wewnętrznych obliczeń podobne jest do pola obliczeniowego. Różnica polega jedynie na tym, że jego wartość nie musi być każdorazowo obliczana, gdy wywoływana jest procedura obsługi zdarzenia *OnCalcFields*. Procedura ta jest uruchamiana zawsze, gdy wyświetlana jest wartość pola, ale ponieważ komponent *ClientDataSet* przechowuje w pamięci podręcznej wartości już obliczonych pól wewnętrznych, wtedy wystarczy ją tylko określić, gdy jest wyświetlana po raz pierwszy lub gdy zostanie zmodyfikowany rekord. W przypadku stosowania pól wewnętrznych obliczeń należy sprawdzić wartość właściwości *State* powiązanego zestawu danych klienta. Jeśli jest to wartość *dsInternalCalc*, wtedy konieczne będzie wykonanie ponownego obliczenia.

Pole agregacji wyświetla statystyki podsumowujące, takie jak suma lub średnia obliczone na rekordach zestawu danych klienta. Weźmy dla przykładu pole *TOTAL_VALUE* tabeli *SALES*. Pole agregacji może służyć do wyświetlenia sumy pola *TOTAL_VALUE* dla każdego klienta. Aby zastosować pola agregacji, zestaw danych klienta musi posiadać indeks, który uporządkuje dane zgodnie z obliczanymi wartościami pola. Na przykład w celu obliczenia sumy pola *TOTAL_VALUE* dla każdego klienta, należy zastosować indeks, który posortuje dane według klienta.

Najczęściej stosowanymi typami nowych pól stałych są: pola obliczeniowe, wyszukiwania oraz agregacji. Każde z nich omówimy w dalszej części rozdziału.

Tworzenie pola obliczeniowego

Dane wyświetlane w polu obliczeniowym określane są przez kod dodany do procedury obsługi zdarzenia *OnCalcFields* zestawu danych. Procedura ta jest wykonywana za każdym razem, gdy zestaw danych musi odświeżyć swoje dane. Ma to miejsce, gdy zestaw danych otwierany jest po raz pierwszy, czytane są rekordy lub zestaw danych przechodzi w tryb edycji. Procedura obsługi zdarzenia *OnCalcFields* jest również wykonywana za każdym razem, gdy w kontrolce bazodanowej wyświetlany jest rekord, ale tylko wtedy, gdy właściwość *AutoCalcFields* ma wartość domyślną *True*.

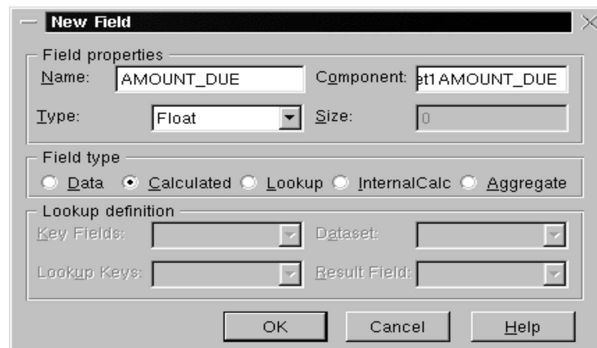
Jeśli procedura obsługi zdarzenia *OnCalcFields* jest wykonywana, wtedy zestaw danych umieszczany jest w jednym z dwóch specjalnych stanów: *dsCalcFields* lub *dsInternalCalc*. Jeśli aktywny jest którykolwiek z nich, to można przypisywać wartości tylko polom

obliczeniowym. Innymi słowy, w procedurze obsługi zdarzenia *OnCalcFields* nie można przypisywać wartości do pól, które związane są z rzeczywistymi polami powiązanego zestawu danych.

Parametr zestawu danych w procedurze obsługi zdarzenia *OnCalcFields* nie wskazuje na bieżący rekord. Zamiast tego wskazuje na rekord powiązany z polem obliczeniowym. Pozwala to zastosować odwołanie do zestawu danych w celu odczytania danych z dowolnego innego pola rekordu i następnie wykonać obliczenie. W celu wykonania obliczenia można również wykorzystać inne dostępne informacje. Można na przykład wykonać na innym zestawie danych polecenie wyszukiwania, przeszukać obiekt *StringList* lub odczytać dane z kontrolki wizualnej. W tym przypadku decyzja należy do Ciebie.

Poniższe kroki opisują, jak utworzyć pole obliczeniowe i wprowadzić do niego dane przy użyciu procedury obsługi zdarzenia *OnCalcFields*. W tym przykładzie zostanie dodane nowe pole obliczeniowe do komponentu *SQLClientDataSet* wcześniej utworzonego projektu. Pole to obliczy należną kwotę sprzedaży w oparciu o wartość pola *TOTAL_VALUE* pomniejszoną o wartość upustu (wyznaczonego przez pomnożenie wartości pól *TOTAL_VALUE* i *DISCOUNT*).

1. Wyświetl moduł danych projektu wykorzystywanego w poprzednich przykładach tego rozdziału.
2. Aby wyświetlić edytor pól komponentu *SQLClientDataSet1*, kliknij go dwukrotnie.
3. Kliknij prawym klawiszem myszy edytor pól i wybierz *New Fields*. Kylix wyświetli okno dialogowe *New Field*.
4. W polu *Name* wpisz *AMOUNT_DUE* i ustaw w polu *Type* wartość *Float*.
5. Następnie wybierz wartość *Calculated* dla opcji *Field type*. Okno dialogowe *New Field* powinno teraz wyglądać jak na rysunku 8.12.



Rysunek 8.12. *Okno dialogowe New Field z dodanym polem obliczeniowym*

6. Aby zamknąć okno dialogowe *New Field*, naciśnij *OK*, a następnie dodaj w edytorze pól nowe pole obliczeniowe. Domyślną nazwą tego pola jest *SQLClientDataSet1AMOUNT_DUE*. Wolno ją zmienić, ale w tym przykładzie może taka pozostać.

7. W oknie *Object Inspector* wybierz pole *SQLClientDataSet1AMOUNT_DUE* i ustaw wartość właściwości *Currency* na *True*.
8. Teraz w oknie *Object Inspector* wybierz komponent *SQLClientDataSet1* i dodaj do niego procedurę obsługi zdarzenia *OnCalcFields*. Wprowadź do niej poniższy kod:

```
procedure TDataModule2.SQLClientDataSet1CalcFields(DataSet:TDataSet);
begin
 SQLClientDataSet1AMOUNT_DUE.Value :=
   SQLClientDataSet1TOTAL_VALUE.Value -
     (SQLClientDataSet1TOTAL_VALUE.Value * SQLClientDataSet1DISCOUNT.Value);
end;
```

9. Zapisz projekt i uruchom go. Za pomocą poziomego paska przewijania przewiń siatkę, tak aby zobaczyć pole *AMOUNT_DUE*. Pole to powinno wyświetlić wyniki obliczeń podobne do tych na rysunku 8.13.

Form1

Zamien na duze litery wartosc pola ORDER_STATUS | Zamien na male litery wartosc pola ORDER_STATUS

PAID	QTY_ORDERED	TOTAL_VALUE	DISCOUNT	ITEM_TYPE	AGED	AMOUNT_DUE
y	10	5000	0.100000001	hardware	1	$4,500.00
y	7	70000	0	hardware	1	$70,000.00
y	3	2000	0	software	93	$2,000.00
y	15	2985	0	software	9	$2,985.00
n	3	60000	0.200000002	hardware		$48,000.00
y	1000	560000	0.200000002	hardware	0	$448,000.00
y	1	0	1	software	0	$0.00
y	2	600.5	0	software	28	$600.50
n	5	20000	0	other	5	$20,000.00
n	150	14850	0.050000000	software	5	$14,107.50
y	20	12582.12	0.100000001	software	11	$11,323.91
y	900	27000	0.300000011	software	21	$18,900.00
n	3	0	1	software		$0.00
y	1	100.02	0	software	1	$100.02
y	1	47.5	0	other	70	$47.50
y	40	399960.5	0.100000001	hardware	24	$359,964.45

Rysunek 8.13. *Pole AMOUNT_DUE wyświetla wyniki obliczeń uzyskanych z procedury obsługi zdarzenia OnCalcFields komponentu SQLClientDataSet*

Tworzenie pól wyszukiwania

W rozdziale 7. pokazano, jak skonfigurować komponent *DBLookupListBox*, aby wyświetlał na liście wartości pola *FULL_NAME* tabeli *EMPLOYEE* w oparciu o wartości pola *SALES_REP* tabeli *SALES*. Oprócz tego wspomnieliśmy, że do zmiany wartości pola *SALES_REP* tabeli *SALES* można użyć komponentu *DBLookupListBox*.

Stałe pola wyszukiwania mogą znaleźć podobne zastosowanie. Pole wyszukiwania na pewno wyświetli powiązane dane z innego zestawu danych. Ponadto gdy pole takie pojawi się w komponencie *DBGrid*, to domyślnie siatka pozwoli użytkownikowi ustawić pole wyszukiwania w trybie edycji, co spowoduje pojawienie się przycisku menu rozwijanego. Gdy użytkownik kliknie ten przycisk, może w celu zmiany wartości pola wybrać odpowiadające mu wartości wyszukiwania. Inaczej mówiąc, pole wyszukiwania wyświetlone w siatce tworzy pole, które zachowuje się podobnie jak komponent *DBLookupComboBox*.

Poniższe kroki opisują proces tworzenia stałego pola wyszukiwania dla pola *SALES_REP* tabeli *SALES* projektu już wcześniej wykorzystywanego w tym rozdziale.

1. Wyświetl moduł danych projektu używanego w poprzednich przykładach.
2. Dodaj do modułu danych kolejny komponent *SQLClientDataSet* z zakładki *dbExpress* palety komponentów. Ustaw wartość właściwości *DBConnection* tego komponentu na *SQLConnection1* i wpisz we właściwości *CommandText* wyrażenie `select EMP_NO, FULL_NAME from EMPLOYEE ORDER BY FULL_NAME`. Na końcu ustaw wartość właściwości *Active* na *True*.
3. Aby otworzyć edytor pól komponentu *SQLClientDataSet1*, kliknij go dwukrotnie.
4. Kliknij prawym klawiszem myszy edytor pól i wybierz *New Fields*. Kylix otworzy okno dialogowe *New Field*.
5. W polu *Name Field* wpisz *EMPLOYEE_NAME*, a następnie ustaw w polu *Type* wartość *String* oraz w polu *Size Field* podaj wartość `24`.
6. Następnie ustaw wartość opcji *Field Type* na *Lookup*. Po wykonaniu tych czynności pola *Key Fields* oraz *Dataset Fields* znajdujące się poniżej sekcji *Lookup definition* powinny być aktywne. Ustaw wartość pola *Key Fields* na *SALES_REP*, natomiast *Dataset* na *SQLClientDataSet2*. Teraz pola *Lookup Keys* oraz *Result Field* powinny się uaktywnić. Ustaw wartość pola *Lookup Keys* na *EMP_NO*, natomiast *Result Field* na *FULL_NAME*. Aby zapisać nowe pole wyszukiwania, kliknij przycisk *OK*.
7. Zapisz projekt i uruchom go. Następnie przewiń siatkę komponentu *DBGrid* w prawo, tak aby pojawiło się pole *EMPLOYEE_NAME*. Aby wejść w tryb edycji pola, kliknij je. Następnie, tak jak to widać na rysunku 8.14, w celu wyświetlenia menu rozwijanego zawierającego nazwiska pracowników z tabeli *EMPLOYEE* kliknij strzałkę rozwijania. Wybierz jedno z nazwisk i przewiń siatkę w drugą stronę, tak aby sprawdzić, czy w polu *SALES_REP* zmienił się numer dla wybranego pracownika.

Rysunek 8.14. *Podczas edycji pola wyszukiwania w komponencie DBGrid automatycznie tworzone jest menu rozwijane z pozycjami tabeli wyszukiwania*

W przykładzie tym wyszukiwanie opiera się na pojedynczym polu. Ściśle mówiąc, pole *SALES_REP* tabeli *SALES* odpowiada polu *EMP_NO* tabeli *EMPLOYEE*. Istnieje możliwość utworzenia pola wyszukiwania, w którym wyszukiwanie opiera się nie tylko na jednym polu, ale na dwóch lub więcej. Aby utworzyć pole wyszukiwania oparte na łączu zawierającym dwa lub więcej pól, należy oddzielić średnikami nazwy odpowiednich pól. Dotyczy to zarówno pola tekstowego *Key Fields*, jak i *Lookup Keys* okna dialogowego *New Field*.

Definiowanie pól agregacji

Pola agregacji dokonują wyliczenia prostych statystyk w oparciu o grupę rekordów. Zaliczają się do nich: suma, średnia, wartość maksymalna i minimalna. Grupami rekordów, na których dokonywane są obliczenia, mogą być wszystkie rekordy wynikowego zestawu albo jeden lub kilka rekordów, które wykorzystują podobne wartości jednego lub kilku poindeksowanych pól. Weźmy dla przykładu tabelę *SALES* stosowaną w poprzednich przykładach tego rozdziału. Można utworzyć pole agregacji, które obliczy sumę wartości pola *TOTAL_VALUE* dla całego wynikowego zestawu. Jeśli utworzymy indeks pola *CUST_NO*, można wtedy również wykreować pole agregacji, które obliczy sumę wartości pola *TOTAL_VALUE* dla każdego klienta.

Pola agregacji nie działają prawidłowo w początkowej wersji Kyliksa. Aby sprawdzić, czy dostępna jest poprawka usuwająca ten błąd, zajrzyj na stronę producenta pod adresem *www.borland.com*. Można również zobaczyć, czy na stronie autora tej książki, pod adresem *http://www.jensendatasystems.com/bka_book.htm*, są informacje dotyczące możliwości uzyskania najnowszych aktualizacji Kyliksa.

Chociaż pola agregacji są przydatne, ich zastosowanie wiąże się z wykonaniem kilku kroków w określonej kolejności. W przeciwnym razie nie będą działać.

1. Aby pogrupować jedno lub kilka pól, utwórz indeks, który te pola obejmuje. Indeks może również dotyczyć innych pól, ale pola, które mają zostać pogrupowane, muszą się znaleźć na jego początku. Aby na przykład indeks dotyczył pola *CUST_NO*, wtedy musi być ono pierwszym jego polem. Zatem zarówno indeks oparty na polu *CUST_NO*, jak i indeks oparty na polach *CUST_NO* oraz *PO_NUMBER* będzie poprawny tak długo, jak pole *CUST_NO* będzie jego pierwszym polem.
2. Utwórz nowe stałe pole agregacji.
3. Przypisz do właściwości *IndexName* tego pola nazwę indeksu, który posłuży grupowaniu.
4. Zdefiniuj wyrażenie agregacji. Mogą być w nim użyte następujące operatory: `sum`, `average`, `count`, `Min` oraz `Max`. Operator może być zastosowany do pojedynczego pola lub w wyrażeniu, które zawiera kilka pól i stałych. Poniżej pokazano przykłady poprawnych wyrażeń agregacji:

```
count( CUST_NO)
average( TOTAL_VALUE - (TOTAL_VALUE * DISCOUNT))
sum(TOTALVALUE) * 100
sum(TOTAL_VALUE) * SUM(DISCOUNT)
```

5. Ustaw poziom grupowania pola agregacji. Poziom grupowania określa ilość pól grupowaną przez indeks. W przypadku indeksu z trzema polami, aby pogrupować pierwsze i drugie pole, należy ustawić wartość grupowania na 2. Natomiast aby utworzyć sumę wartości pola *TOTAL_VALUE* według wartości pola *CUST_NO*, należy ustawić wartość *GroupingLevel* na 1.
6. Ustaw wartość właściwości *Active* pola agregacji na *True*.
7. Ustaw wartość właściwości *AggregatesActive* zestawu danych, do którego należy pole agregacji, na *True*.
8. Uaktywnij zestaw danych, ustawiając wartość jej właściwości *Active* na *True* lub wywołując po uruchomieniu aplikacji metodę `Open`.

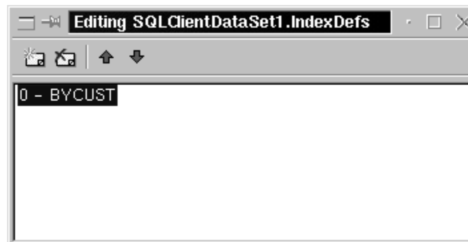
Poniższe kroki opisują proces tworzenia pola agregacji wyświetlającego sumę wartości pola *TOTAL_VALUE* według wartości pola *CUST_NO*.

1. Otwórz moduł danych projektu wykorzystywanego w poprzednich przykładach rozdziału.
2. Wybierz komponent *SQLClientDataSet1*.
3. W oknie *Object Inspector* wybierz właściwość *IndexDefs* komponentu *SQLClientDataSet1* i następnie, aby wyświetlić edytor definicji indeksu (patrz rysunek 8.15), kliknij przycisk wielokropka.

Rysunek 8.15.
Edytor właściwości IndexDefs

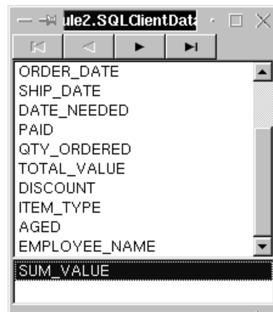
4. Kliknij przycisk *Add New*. Po zaznaczeniu właściwości *IndexDefs* w oknie *Object Inspector* ustaw wartość właściwości *Name* na *BYCUST* (patrz rysunek 8.16), natomiast właściwości *Fields* na *CUST_NO*. Zamknij edytor *FieldDefs*.

Rysunek 8.16.
Edytor właściwości IndexDefs



5. Wybierz ponownie komponent *SQLClientDataSet1* i ustaw wartość właściwości *IndexName* na *BYCUST*.
6. Aby otworzyć edytor pól, kliknij dwukrotnie komponent *SQLClientDataSet1*. Aby otworzyć okno dialogowe *New Field*, naciśnij klawisze *Ctrl+N*.

7. W polu *Name Field* wpisz `SUM_VALUE` i następnie ustaw opcję *Field Type* na *Aggregate*. W celu zamknięcia okna dialogowego *New Field* naciśnij przycisk *OK*.
8. Utworzone pole agregacji pojawi się w edytorze pól. W przeciwieństwie do innych pól stałych wszystkie pola agregacji wyświetlane są na oddzielnej liście w dolnej części edytora, tak jak to widać na rysunku 8.17.



Rysunek 8.17. *Edytor pól*

9. Zaznacz w edytorze pól pole agregacji *SUM_VALUE* i w oknie *Object Inspector* ustaw wartość właściwości *IndexName* na *BYCUST*. Następnie ustaw wartość właściwości *GroupingLevel* pola na `1`, a właściwości *Expression* na `sum(TOTAL_VALUES)`. Na samym końcu ustaw wartość właściwości *Currency* pola oraz właściwości *Active* na *True*.
10. Wybierz ponownie komponent *SQLClientDataSet1* i ustaw wartość jego właściwości *AggregatesActive* oraz właściwości *Active* na *True*.
11. Umieść komponent *Label* z zakładki *Standard* palety komponentów oraz komponent *DBEdit* z zakładki *Data Controls* palety komponentów po prawej stronie dwóch przycisków znajdujących się w górnym panelu. Ustaw wartość właściwości *Caption* komponentu *Label* na *Suma wartości pola TOTAL_VALUE wg klienta*. Następnie ustaw wartość właściwości *DataSource* komponentu *DBEdit* na *DataModule2.DataSource1*, a właściwości *DataField* na *SUM_VALUE*.
12. Zapisz zmiany w projekcie i uruchom go.

Po uruchomieniu główny formularz powinien wyglądać podobnie jak na rysunku 8.18. Podczas przejścia od jednego rekordu klienta do następnego zmienia się wartość wyświetlana w polu agregacji. Ponadto w przypadku edycji danych pola *TOTAL_VALUE* dla określonego klienta po zatwierdzeniu wprowadzonej wartości wartość pola agregacji również się zmieni.

Ponieważ pola agregacji nie działają prawidłowo w początkowej wersji Kyliksa, również ten przykład nie zadziała. Mimo że w czasie nawigacji po rekordach klientów zmienia się wartość, nie ma to nic wspólnego z polem *TOTAL_VALUE*, a powinno. Również w zależności od wersji poprawki firmy Borland związanej z polami agregacji kod i poszczególne kroki przykładu mogą wymagać pewnych modyfikacji. Aby uzyskać więcej informacji, zajrzyj do pomocy lub dokumentacji dostarczonej z wersjami Kyliksa o numeracji 1.x.

Rysunek 8.18. *Pole agregacji zmienia się automatycznie w przypadku przechodzenia między rekordami klientów i ich edycji*

Form1

Zamien na duze litery wartosc pola ORDER_STATUS | Zamien na male litery wartosc pola ORDER_STATUS

Suma wartosci pola TOTAL_VALUE wg klienta 5070000

PO_NUMBER	CUST_NO	SALES_REP	ORDER_STATUS	ORDER_DATE	SHIP_DATE
V9324200	1001	72	shipped	08/09/1993	08/09/1993
V9320630	1001	127	open	12/12/1993	
V9420099	1001	127	open	01/17/1994	
V9427029	1001	127	shipped	02/07/1994	02/10/1994
V9324320	1001	127	shipped	08/16/1993	08/16/1993
V9336100	1002	11	waiting	12/27/1993	01/01/1994
V9333005	1002	11	shipped	02/03/1993	03/03/1993
V9333006	1002	11	shipped	04/27/1993	05/02/1993
V9346200	1003	11	waiting	12/31/1993	
V9345200	1003	11	shipped	11/11/1993	12/02/1993
V9345139	1003	127	shipped	09/09/1993	09/20/1993
V91E0210	1004	11	shipped	03/04/1991	03/05/1991
V92E0340	1004	11	shipped	10/15/1992	10/16/1992
V93H0030	1005	118	open	12/12/1993	
V94H0079	1005	61	open	02/13/1994	

W następnym rozdziale omówimy zagadnienia dotyczące bezpośredniej pracy z zestawami danych.