**Основы ООП в Delphi**

*Предпосылки возникновения ООП*

Рост сложности кода:

* Следование/ветвление/цикл
* Процедуры
* Модули
* …

Рост сложности данных:

* Скалярные типы
* Агрегированные (составные) типы
* …

ООП – парадигма (идея)

Программа состоит из объектов, которые обмениваются сообщениями

*Эволюция ООП*

Type

TButton = record

X, Y, W, H: Integer;

…

End;

Procedure SetPos(var Button: TButton; X, Y: Integer);

Type

TListBox = record

X, Y, W, H: Integer;

Items: array of String;

…

End;

Procedure SetPos(var ListBox: TListBox; X, Y: Integer);

…

Procedure Button\_SetPos(var Button: TButton; X, Y: Integer);

Procedure ListBox\_SetPos(var ListBox: TListBox; X, Y: Integer);

Но вот идея:

Type

TControl = record

X, Y, W, H: Integer;

Visible: Boolean;

End;

TButton = record

ControlData: TControl;

…

End;

TListBox = record

ControlData: TControl;

…

End;

Procedure Control\_SetPos(var Control: TControl; X, Y: Integer);

Control\_SetPos(Button.ControlData, 100, 200);

Button\_Draw(Button)

«Три кита» ООП:

* Инкапсуляция
* Наследование
* Полиморфизм

Инкапсуляция

Объединение данных и кода, который их обрабатывает

Наследование

Возможность создавать новые типы данных путем расширения существующих

Полиморфизм

Возможность использовать различные типы данных взаимозаменяемо

Type

TSetPosProc = procedure (var Control; …);

TDrawProc = procedure (var Control);

…

TButton = record

ControlData: TControl;

Caption: String;

SetPos: TSetPosProc;

Draw: TDrawProc;

End;

MyButton.SetPos(MyButton, 100, 200);

MyButton.Draw(MyButton);

Было:

Control\_SetPos(Button.ControlData, 100, 200);

Button\_Draw(Button)

ООП не привязан к ЯП:

* Писать в ООП-стиле можно на любом языке программирования.
* Но проще, если в ЯП есть поддержка ООП (на уровне синтаксиса)

Type

TControl = class

X, Y, W, H: Integer;

Visible: Boolean;

Procedure SetPos(X, Y:Integer);

Procedure Draw;

…

End;

Все объекты в Delphi выделяются в динамической памяти.

* Переменная хранит адрес объекта

В теле каждого метода доступна переменная Self.

* Содержит адрес объекта, у которого был вызван метод.
* (Кроме методов класса)

**Наследование**

Класс может наследоваться от другого класса

* При этом наследуемый класс будет содержать все члены родительского.
* Но сможет добавить новые…
* … или изменить видимость существующих
* Если не указать родительский класс, им будет TObject

Type

TShape = class

FColor: TColor;

End;

TRectangle = class(TShape)

FWidth, FHeight: Real;

Procedure Draw;

End;

(TRectangle теперь как TShape, но + FWidth, FHeight и Procedure Draw;

Членам класса могут задаваться различные уровни видимости:

* Private
* Protected
* Public
* Published

Private

* Внутри методов этого же класса

Protected

* Внутри методов этого же класса…
* … или наследуемых от него.

Public

* Везде.

В Delphi есть некоторые особенности:

* Private-члены доступны всем внутри модуля, в котором объявлен класс
* Есть published-видимость (используется для интеграции со средой);
* По умолчанию – public/published

*Члены класса*

Члены класса:

* Поля
* Методы
* Свойства/события

Могут принадлежать:

* Экземплярам класса (объектам)
* Самому классу

Реализация метода

Определяющее описание (реализация) метода:

* По синтаксису аналогично обычным подпрограммам/функциям
* Но используют составное имя

Procedure TMyClass.SomeMethod(…);

Begin

…

End;

*Обращение к унаследованным методам*

Для обращения к унаследованным методам используется ключевое слово inherited.

// Вызов унаследованного метода Draw

inherited Draw(…);

// Вызов метода с тем же именем,

// Что и вызывающий, и с теми же параметрами

inherited

Методы могут быть:

* Статическими
* Виртуальными
* Динамическими
* Методами класса

Особо выделяют:

* Конструкторы
* Деструкторы

*Конструкторы*

Вводятся ключевым словом constructor. (Крч для инициализации, при создании)

* Могут иметь произвольные имена, но общепринятое – Create

Создание объекта:

Var

MyObj: TShape;

…

MyObj := TShape.Create(…);

*Деструкторы*

Вводятся ключевым словом destructor. (Крч для free памяти при удалении)

* Могут иметь произвольные имена, но общепринятое – Destroy. (Напрямую не вызывают Destroy (FPen.Free; inherited Destroy), т.к. FPen может быть nil???) (FPen – пример)

Поля создаваемого объекта инициализируются нулями.

Если конструктор выбрасывает исключение, создание объекта прерывается.

* Деструктор при этом будет вызван…
* … но объект при этом всё еще инициализирован не полностью (некоторые поля содержат нули)

У класса TObject реализован метод Free:

Procedure TObject.Free;

Begin

If Self <> nil then

Destroy;

End;

*Процедура FreeAndNil*

Частая ошибка – двойное освобождение памяти.

* Например, разрушение уже разрушенного объекта

В модуле SysUtils объявлена процедура FreeAndNil:

* Вызывает метод Free;
* Записывает в переменную-объект nil

**Свойства**

Свойство – способ доступа к внутреннему состоянию объекта, имитирующий поведение переменной.

* Внутреннее состояние объекта задается значениями его полей.

Свойство может быть реализовано.

* Прямым доступом к какому-либо полю
* В виде метода, вызываемого при обращении к свойству

Property Color: TColor read GetColor write SetColor;

Для прогера:  
MyColorClass.Color := clRed 🡪 вызовется SetColor

*Свойства-массивы*

Property Items[Index: Integer]: TObject read GetObject write SetObject

*Индексированные свойства*

Property Left: LongInt index 0 read GetCoordinate write SetCoordinate

**События**

Событие – сообщение, возникающее в различных местах программы при выполнении определенных условий:

* Нажатие на кнопку
* Выбор элемента в списке
* И т.д.

Обработчик события – ~~процедура~~ метод, определяющая реакцию программы на то или иное событие

Один из способов реализации инверсии управления.

Представляют собой свойства процедурного типа.

* Точнее – указатель на метод.

TOnClickEvent = procedure(X, Y: Integer) of object;

TMyClass = class

Private

FOnClick: TOnClickEvent;

Public

Property OnClick: TOnClickEvent read FOnClick write FOnClick;

End;

…

Procedure TMyClass.SomeMethid;

Begin

If Assigned(FOnClick) then

FOnClick(10, 20);

End;

**Исключения**

Исключение – ошибка времени выполнения или другая проблема, приводящая к невозможности или бессмысленности дальнейшего выполнения алгоритма.

* Языки программирования предоставляют специальный синтаксис для обработки исключительных ситуаций

В Delphi исключения – объекты классов, унаследованных от класса Exception

* Имеется целая иерархия стандартных классов-исключений для типовых случаев

*raise*

type

EMyException = class(EInvalidOp);

…

If X >= 0 then

Result := Sqrt(X)

Else

Raise EMyException.Create(‘Negative value.);

При возникновении исключения:

* Выполнение текущего участка кода прерывается
* Управление передается в ближайший обработчик
* Если в текущей подпрограмме нет подходящих обработчиков:
  + Они ищутся в подпрограмме, вызвавшей текущую подпрограмму

Грубая ошибка!

Обработка исключений простым **try…except** без фильтрации.

Проблема:

* Исключение может быть связано не с теми проблемами, которых ожидает программист

Решение:

* Использовать фильтрацию исключений

Пример фильтрации:

Try

// Codic vsyakiy

Except

onEConvertError do

// obrabotka EConvertError

On EZeroDivide do

// …

Else

// Obrabotka ostalnyh oshibok

End;

Try

// Код, который может выбросить исключение или выйти из блока досрочно

Finally

// Код, который нужно выполнить при выходе из блока всегда

End;

*Повторное выбрасывание исключения*

При обработке исключения можно:

* Выполнив обработку, повторно выбросить то же самое исключение
* Выполнив обработку, выбросить другое исключение

Try

// Code

Except

// Obrabotka

Raise EAnotherException.Create(‘…’);

End;

*Главная ошибка начинающих!!!*

Злоупотребление ООП.

* ООП как идея подходит не для всех задач и/или частей программ.
* Признаки плохого «ООП»:
  + Объекты без методов (только данные)
  + Объекты без данных (только методы)
  + И др.
* ООП как набор возможностей языка предполагает накладные расходы:
  + Как правило, код с классами/объектами менее эффективен, чем без них.

**Основы GUI-приложений в Delphi**

Библиотеки для разработки GUI-приложений в Delphi:

* VCL – Visual Component Library;
* FireMonkey

Структура проекта GUI-приложения

* Файл проекта \*.dpr
* Формы \*.dfm 🡨🡪 Модули форм \*.pas
* Другие модули \*.pas

Компонент – элемент программы:

* Может быть размещен на форме
* В общем случае не имеет визуального представления

Контрол (элемент управления) компонент:

* Имеет визуальное представление
* Используется для взаимодействия пользователя с программой

*Ошибка №0*

Использование имён компонентов (и модулей) по умолчанию

Проблема:

* Имя не отражает назначения.
* В сложных формах с большим количеством компонентов сложнее ориентироваться.

Решение:

* Давать компонентам осмысленные имена
* Добавлять к имени префикс обозначающий тип компонента

*Событийно-ориентированное программирование*

СОП – способ построения программы, при котором в коде явным образом выделяется главный цикл приложения:

* Выборка события
* Обработка события

*Ошибка №1*

Обращение к конкретной форме из обработчиков событий.

Проблема:

* Если создавать вторую форму того же класса, она будет реагировать на события, изменяя первую форму

Решение:

* Обращаться к форме через переменную Self.
* Переменную Self можно не указывать

*Ошибка №2*

Создание отдельных обработчиков для событий от однотипных контролов.

Проблема:

* Дублирование кода

Решение:

* Использовать параметр Sender

*Обработка событий: Sender*

В обработчики событий передается параметр Sender.

* Значение параметра – объект, на котором возникло событие.
* Позволяет реализовать одинаковое поведение на различных элементах управления

Параметр Sender имеет тип TObject

* Как правило, необходимо приводить его к нужному типу.
* Два способа:
  + TButton(Sender)
  + Sender as TButton

**Программы со сложным UI размещение контролов**

Свойство Align

* Задает способ автоматического позиционирования элемента.
* Позволяет реагировать на изменение размеров окна без необходимости написания кода

Элемент управления TPanel

* Может использоваться для группировки других элементов

**Программы со сложным UI: разделение ответственностей**

*Ошибка №3*

Хранение данных программы в полях контролов.

Проблема:

* При смене используемого контрола может потребоваться изменение структур данных.

Решение:

* Единственное место хранения данных – модель (или хотя бы поля формы).
* Контролы только занимаются их отображением

*Single responsibility principle*

Каждый модуль/класс должен

* Отвечать за одну часть функциональности программы
* Полностью ее инкапсулировать (реализовывать)

Все сервисы (интерфейс) должны иметь отношение только к этой части функциональности

*MVC*

**MVC** – парадигма, в основу которой положена идея разделения кода программы на:

* Модель (model)
* Представление (view)
* Контроллер (controller)

Модель – бизнес логика:

* Данные, относящиеся к предметной области
* Правила их обработки

Представление – способ отображения модели пользователю

Контроллер – реакция на внешние события

Главная идея

Обработка данных программой не должна зависеть от способа их отображения пользователю

*Ошибка №4*

Описание бизнес-логики в обработчиках событий

Проблема:

* Бизнес-логика (модель) привязывается к обработке событий (представление/контроллер).
* При изменении пользовательского интерфейса придется переписывать много кода

Решение:

* Дополнительные процедуры, TActionList

Хороший UI должен соответствовать ожиданиям пользователя

Простые и понятные метафоры

* В крайнем случае – привычные

Различные способы взаимодействия:

* Горячие клавиши
* Экранные кнопки
* Меню
* И т.д.

Визуальные подсказки:

* Иконки
* Текстовые подсказки
* Группировка элементов управления

*TImageList*

Позволяет управлять набором изображений

* Изображения одинакового размера
* Чаще всего используется для управления иконками
* Формирует большое изображение, состоящее из исходных (тайлы)
* Каждое изображение получает индекс (начиная с 0)

*TActionList*

Группирует действия, доступные пользователю

С каждым действием связывается информация:

* Отображаемый текст (Caption)
* Доступность (Enabled)
* Индекс изображения (ImageIndex)
* Горячие клавиши (ShortCut)
* И т.д.

У каждого действия есть свой обработчик

Событие OnUpdate позволяет централизованно обновлять действия

* Есть у самого TActionList и у отдельных действий

*Использование неподходящих контролов*

Использование TImage для построения изображений:

* TImage – для работы с изображениями в различных форматах:
  + Вместе с ним в программу «тянется» код, отвечающий за поддержку работы с ними
* TPaintBox – для рисования (Не успел дописать)

Использование TStringGrid для «массивов записей»:

* Гриды заточены на отображение табличных данных (двумерные массивы);
* Для «массивов записей» лучше использовать TListView и его аналоги:
  + Предусмотрена сортировка по столбцам
  + Различные режимы отображения
  + Экономия памяти и процессорного времени при работе с большими объемами данных