КЕМЕРОВСКИЙ ИНСТИТУТ (филиал)
РОССИЙСКИЙ ГОСУДАРСТВЕННЫЙ ТОРГОВО-ЭКОНОМИЧЕСКИЙ УНИВЕРСИТЕТ
КАФЕДРА ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ И ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

Информатика и программирование

Лебедева Т.Ф.

ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ ПОЛОЖЕНИЯ

Объектно-ориентированное программирование (ООП) представляет собой следующий этап развития современных концепций построения языков программирования. Здесь получили дальнейшее развитие принципы структурного программирования - структуризация программ и данных, модульность и т. д.

В основе ООП лежит понятие объекта (object), сочетающего в себе данные и действия над ними. Объект в некотором роде похож на стандартный тип-запись (record), но включает в себя не только поля данных, но также и подпрограммы для обработки этих данных, называемые методами. Таким образом, в объекте сосредоточены его свойства и поведение.

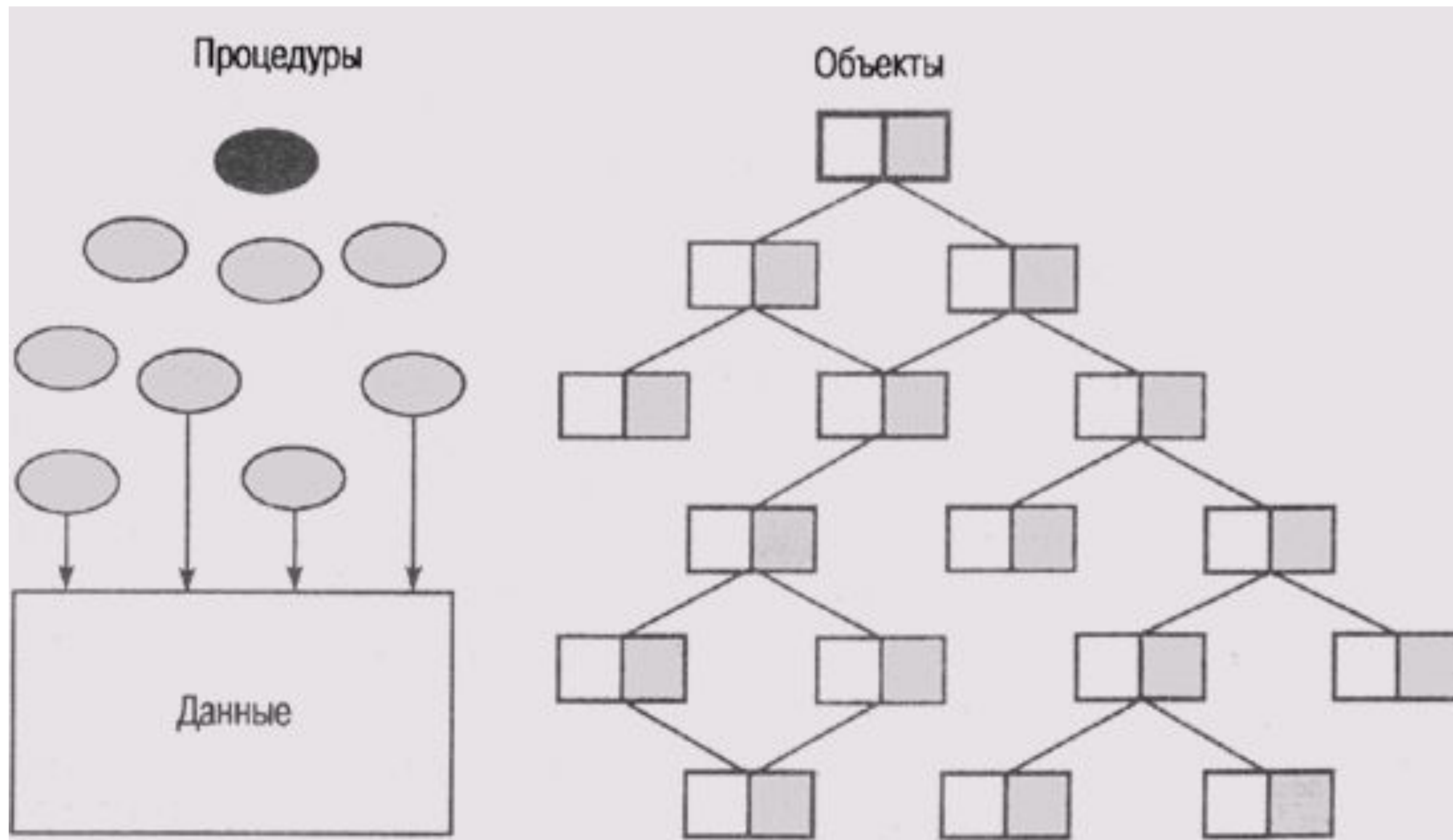


Рис.1. Процедурно- и объектно-ориентированные парадигмы программирования. Незаполненные фигуры представляют данные, а фигуры с заливкой—процедуры

Введение нового типа данных потребовало пересмотреть некоторые концепции языка Паскаль: ввести новые понятия, как, например, инкапсуляция, наследование, полиморфизм и виртуальность, новые зарезервированные слова (constructor, destructor, inherited, object, private, public, virtual), изменить уже существующие подпрограммы (подпрограммы New и Dispose).

ООП характеризуется тремя основными свойствами: инкапсуляция (encapsulation), наследование (inheritance) и полиморфизм (polymorphism).

Инкапсуляция означает упоминавшееся выше объединение в одном объекте данных и действий над ними. Примером может служить перемещаемый по экрану отрезок, определяемый координатами своих концов (данные), и процедурой, обеспечивающей это перемещение (метод).

Наследование позволяет создавать иерархию объектов, начиная с некоторого простого первоначального (предка) и кончая более сложными, но включающими (наследующими) свойства предшествующих элементов (потомки). Эта иерархия в общем случае может иметь довольно сложную древовидную структуру. Каждый потомок несет в себе характеристики своего предка (содержит те же данные и методы), а также обладает собственными характеристиками. При этом наследуемые данные и методы описывать у потомка нет необходимости.

В качестве такой иерархии можно рассмотреть точку на экране дисплея, задаваемую своими координатами (предок), отрезок, задаваемый координатами двух точек - его концов (потомок точки), перемещаемый отрезок, задаваемый координатами своих концов и процедурой, обеспечивающей его перемещение (потомок неперемещаемого отрезка) и т. д.

Полиморфизм означает, что для различных родственных объектов можно задать единый класс действий (например, перемещение по экрану любой геометрической фигуры). Затем для каждого конкретного объекта составляется своя подпрограмма, выполняющая это действие непосредственно для данного объекта (естественно, что перемещение по экрану точки отличается от перемещения отрезка, а перемещение отрезка, в свою очередь, отличается от перемещения многоугольника и т. д.), причем все эти подпрограммы могут иметь одно и то же имя. Когда потребуется перемещать конкретную фигуру, будет выбрана из всего класса соответствующая подпрограмма.

Гради Буч: *«Объектно-ориентированное программирование (ООП) – это методология программирования, которая основана на представлении программы в виде совокупности объектов, каждый из которых является реализацией (экземпляром) определенного класса (типа особого вида), а классы образуют иерархию на принципах наследуемости».*

Как следует из определения, ООП в отличие от процедурного программирования, базируется на объектной декомпозиции предметной области программы.

ООП обладает рядом *преимуществ* при создании больших программ. В частности, к ним можно отнести:

- использование более естественных с точки зрения повседневной практики понятий, простота введения новых понятий;
- некоторое сокращение размера программ за счет того, что повторяющиеся (наследуемые) свойства и действия можно не описывать многократно, как это делается при использовании подпрограмм; кроме того, использование динамических объектов позволяет более эффективно использовать оперативную память;
- возможность создания библиотеки объектов;

- сравнительно простая возможность внесения изменений в программу без изменения уже написанных частей, а в ряде случаев и без перекомпиляции этих написанных и уже скомпилированных частей, используя свойства наследования и полиморфизма;
- возможность написания подпрограмм с различными наборами формальных параметров, но имеющих одно и то же имя, используя свойство полиморфизма;
- более четкая локализация свойств и поведения объекта в одном месте (используется свойство инкапсуляции), позволяющая проще разбираться со структурой программы, отлаживать ее, находить ошибки;
- возможность разделения доступа к различным объектам программы и т. д.

Однако следует иметь в виду, что ООП обладает и рядом *недостатков* и эффективно не во всех случаях.

1) Как правило, использование ООП приводит к уменьшению быстродействия программы, особенно в тех случаях, когда используются виртуальные методы
2) Неэффективно ООП применительно к небольшим программам, поэтому его можно рекомендовать при создании больших программ, а лучше даже класса программ (как, например, создание интерактивных программ с использованием Turbo Vision, где основой является ООП).
3) Можно, по-видимому, даже сказать, что ООП скорее *не упрощает саму программу, а упрощает технологию ее создания.*

Гради Буч выделяет две разновидности декомпозиции: алгоритмическую (поддерживаемую структурными методами) и объектно-ориентированную, основанную на разделении по объектам.

На практике рекомендуется применять обе разновидности декомпозиции:

- При создании крупных проектов целесообразно сначала применить объектно-ориентированный подход для создании общей иерархии объектов, отражающей сущность программируемой задачи,
- а затем использовать алгоритмическую декомпозицию на модули для упрощения разработки и сопровождения разрабатываемого программного комплекса

Объектная декомпозиция

Объектной декомпозицией называют процесс представления предметной области задачи в виде совокупности функциональных элементов (объектов), обменивающихся в процессе выполнения программы входными воздействиями (*сообщениями)*.

Каждый выделяемый объект предметной области отвечает за выполнение некоторых действий, зависящих от полученных сообщений и параметров самого объекта.

Гради Буч дает следующее определение объекта:

«Объект - это мыслимая или реальная сущность, обладающая характерным поведением и отличительными характеристиками и являющаяся важной в предметной области. Каждый объект имеет состояние, обладает четко определенным поведением и уникальной идентичностью».

Объект – это тип данных, который включает не только поля данных объекта, но и подпрограммы (процедуры и функции) для их обработки, называемые методами.

Объектная декомпозиция

Алан Кей в свое время вывел пять основных черт языка Smalltalk — первого удачного ОО языка:

Все является объектом. Объект как хранит информацию, так и способен ее преобразовывать. В принципе любой элемент решаемой задачи (дом, собака, услуга, химическая реакция, город, космический корабль и т. д.) может представлять собой объект. Объект можно представить себе как швейцарский нож: он является набором различных ножей и «открывашек» (хранение), но в то же самое время им мы можем резать или открывать что-либо (преобразование).

Программа — совокупность объектов, указывающих друг другу что делать. Для обращения к одному объекту другой объект «посылает ему сообщение». Как вариант возможно и «ответное сообщение».

Каждый объект имеет свою собственную «память» состоящую из других объектов. Таким образом программист может скрыть сложность программы за довольно простыми объектами. К примеру, дом (достаточно сложный объект) состоит из дверей, комнат, окон, проводки и отопления. Дверь, в свою очередь, может состоять из собственно двери, ручки, замка и петель. **У каждого объекта есть тип.** Иногда тип называют еще и классом. Класс (тип) определяет какие сообщения объекты могут посылать друг другу. Например, аккумуляторная батарея может передавать электролампе ток, а вот момент или физическое усилие - нет.

Все объекты одного типа могут получать одинаковые сообщения. К примеру у нас есть 2 объекта: синяя и красная кружки. Обе разные по форме и материалу. Но из обеих мы можем пить (или не пить, если они пустые). В данном случае кружка — это тип объекта.

Самое лаконичное описание объекта предложил *Буч*: «Объект обладает *состоянием*, *поведением* и *индивидуальностью*».

Объект характеризуется как совокупностью всех своих свойств (например, для животных – это наличие головы, хвоста, ушей, глаз и т. д.) и их текущих значений (голова – большая, уши – длинные, глаза – желтые и т.д.) так и совокупностью допустимых для данного объекта действий (умение есть, дышать, бегать и т.д.).

Указанное объединение в едином объекте как «материальных» составных частей (голова, уши, лапы и т.д.), так и действий, манипулирующих этими частями (действие «бежать» быстро перемещает лапы) называется *инкапсуляцией.*

Совокупность значений параметров объекта называют его *состоянием*, а совокупность реакций на получаемые сообщения - *поведением.*

Состояние (state) - совокупный результат поведения объекта: одно из стабильных условий, в которых объект может существовать, охарактеризованных количественно; в любой момент времени состояние объекта включает в себя перечень (обычно статический) свойств объекта и текущие значения (обычно динамические) этих свойств .

Поведение

Для каждого объекта существует определенный набор действий, которые с ним можно произвести. Например, возможные действия с некоторым файлом операционной системы ПК (объектом):

- создать;
- открыть;
- читать из файла;
- писать в файл;
- закрыть;
- удалить.

Результат выполнения действий зависит от состояния объекта на момент совершения действия, т.е. нельзя, например, удалить файл, если он открыт кем-либо (заблокирован).

В то же время действия могут менять внутреннее состояние объекта - при открытии или закрытии файла свойство "открыт" принимает значения "да" или "нет", соответственно.

Поведение (behavior) - действия и реакции объекта, выраженные в терминах передачи сообщений и изменения состояния; видимая извне и воспроизводимая активность объекта.

Программа, написанная с использованием ООП, обычно состоит из множества объектов, и все эти объекты взаимодействуют между собой. Обычно говорят, что взаимодействие между объектами в программе происходит посредством передачи сообщений между ними.

Параметры состояния и элементы поведения объектов определяются условием задачи.

В процессе решения задачи объект, получив некоторое сообщение, выполняет заранее определенные действия, например, может изменить собственное состояние, выполнить некоторые вычисления, нарисовать окно или график и, в свою очередь, сформировать сообщения другим объектам.

Таким образом, процессом решения задачи управляет последовательность сообщений.

Передавая эти сообщения от объекта к объекту, программа выполняет необходимые действия.

Классы и объекты-переменные

В программе для представления объектов предметной области используют переменные специальных типов - классы.

Класс - это структурный тип данных, который включает описание полей данных, а также процедур и функций, работающих с этими полями данных.

Применительно к классам такие процедуры и функции получили название *методов.*

Поля, описанные в классе, используют для хранения составляющих состояния или атрибутов объекта. Например, если объект Функция должен хранить номер функции, то реализующий его класс должен содержать соответствующее поле.

Класс изображается в виде прямоугольника, состоящего из трех частей. В верхней части помещается название класса, в средней - свойства объектов класса, в нижней - действия, которые можно выполнять с объектами данного класса (методы).

В двух словах, *класс* - это тип данных, а *объект* - экземпляр типа класс. "Кружка" - это класс (тип). А уж которая, - синяя или красная, - это два разных объекта (экземпляра), типа "кружка"

Каждый класс также может иметь специальные методы, которые автоматически вызываются при создании и уничтожении объектов этого класса:

- конструктор (constructor) - выполняется при создании объектов;
- деструктор (destructor) - выполняется при уничтожении объектов.

Имя объекта	Имя класса	Имя объекта
Состояние	Поля	Значения
Поведение	Методы	Методы
Объект предметной области	Класс	Объект-переменная

Рис.2 Соответствие объектов предметной области, классам и объектам-переменным

Переменные типа класса также обычно называют объектами. На рис. 2 показана связь объектов предметной области, классов и объектов-переменных.

Согласно общим правилам языка программирования объект-переменная должен быть:

- *создан* - для него должна быть выделена память;
- *инициализирован* - полям объекта должны быть присвоены значения;
- *уничтожен* - память, выделенная под объект, должна быть освобождена.

Как нам теперь использовать объекты в языках, использующих различные объектные модели?

C++: если у нас есть класс *MyClass* с методом *MyMethod*, мы можем написать:

MyClass Obj;
Obj.MyMethod();

и получить объект класса *MyClass* с именем *Obj*. Память для этого объекта обычно выделяется в стеке, и вы можете сразу начать использовать объект, как это сделано во второй строке.

Также возможно выделить память для объекта в куче и оперировать указателем на объект:

*MyClass *obj = new MyClass();*
obj->MyMethod();
delete obj;//освобождаем память

Java: подобная инструкция выделяет только место для хэндла объекта, а не для самого объекта:

MyClass Obj;
Obj = **new** MyClass();
Obj.MyMethod();

Прежде чем использовать объект, вы должны вызвать "new" для выделения под него памяти. Конечно, вы можете объявить и проинициализировать объект в одном предложении, избегая использования неинициализированных объектных хэндлов:

```
MyClass Obj = new MyClass();
 Obj.MyMethod();
```

Object Pascal: использует подобный подход, но требует отдельных предложений для объявления и инициализации:

```
var
  Obj: MyClass;
begin
  Obj := MyClass.Create;
  Obj.MyMethod;
```

В **C#** работа с объектами классов и структур внешне выглядит похоже:

```
MyClass obj1 = new MyClass();
obj1.Method1();
MyStruct1 obj2 = new MyStruct("Hello!");
obj2.Method2();
MyStruct2 obj3;
obj3.Method3();
```

Наследование

Наследование (inheritance) - это отношение между классами, при котором класс использует структуру или поведение другого класса (одиночное наследование), или других (множественное наследование) классов. Наследование вводит иерархию "общее/частное", в которой подкласс наследует от одного или нескольких более общих суперклассов. Подклассы обычно дополняют или переопределяют унаследованную структуру и поведение.

В качестве примера можно рассмотреть задачу, в которой необходимо реализовать классы "Легковой автомобиль" и "Грузовой автомобиль". Очевидно, эти два класса имеют общую функциональность. Так, оба они имеют 4 колеса, двигатель, могут перемещаться и т.д. Всеми этими свойствами обладает любой автомобиль, независимо от того, грузовой он или легковой, 5- или 12-местный. Разумно вынести эти общие свойства и функциональность в отдельный класс, например, "Автомобиль" и наследовать от него классы "Легковой автомобиль" и "Грузовой автомобиль", чтобы избежать повторного написания одного и того же кода в разных классах (рис.3).

Рис.3 Одиночное наследование

В рассмотренном примере применено одиночное наследование. Некоторый класс также может наследовать свойства и поведение сразу нескольких классов. Наиболее популярным примером применения множественного наследования является проектирование системы учета товаров в зоомагазине.

Все животные в зоомагазине являются наследниками класса "Животное", а также наследниками класса "Товар". Т.е. все они имеют возраст, нуждаются в пище и воде и в то же время имеют цену и могут быть проданы.

Множественное наследование на диаграмме изображается точно так же, как одиночное, за исключением того, что линии наследования соединяют класс-потомок сразу с несколькими суперклассами.

Не все объектно-ориентированные языки программирования содержат языковые конструкции для описания множественного наследования.

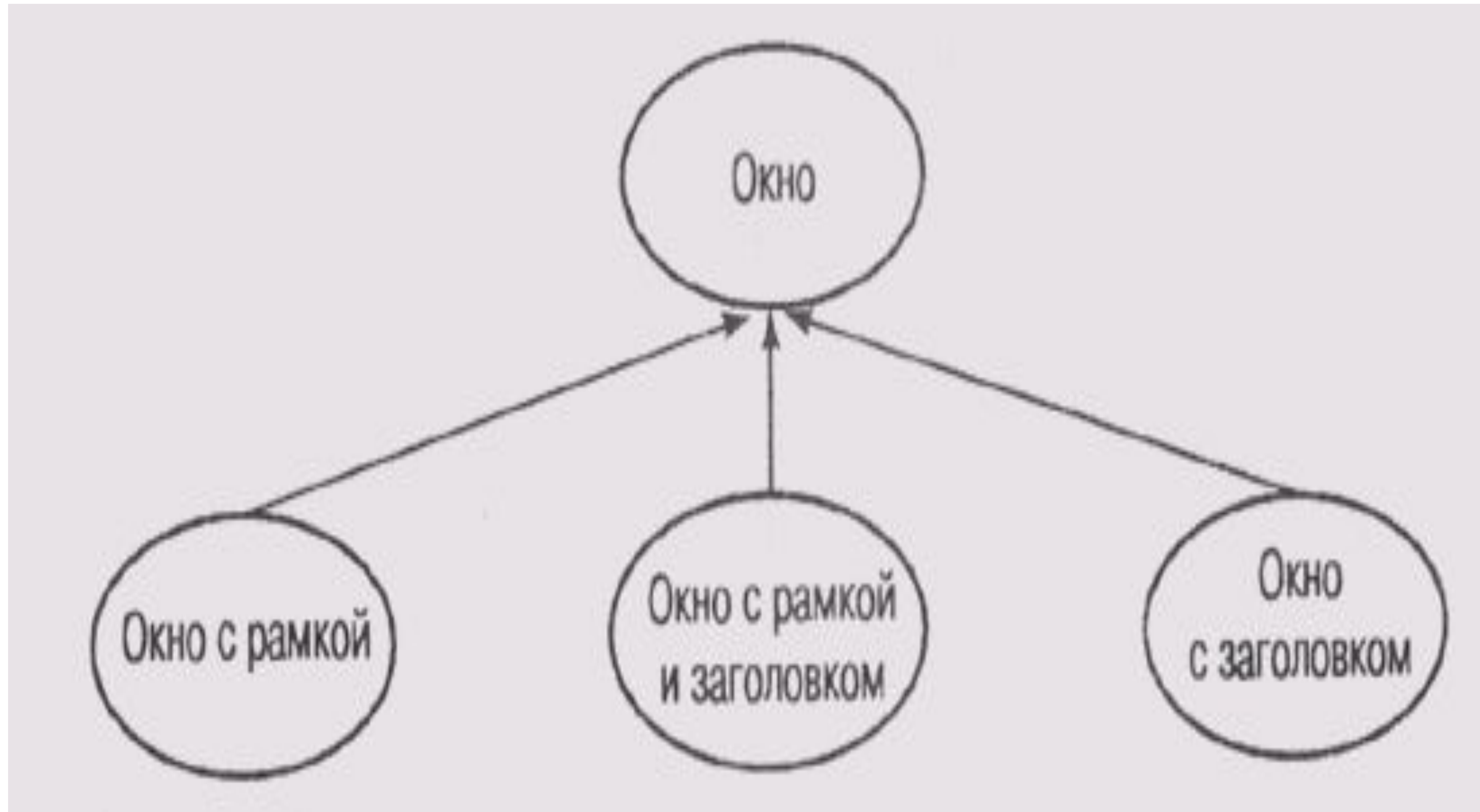


Рис.4 Иерархическая система классов окон

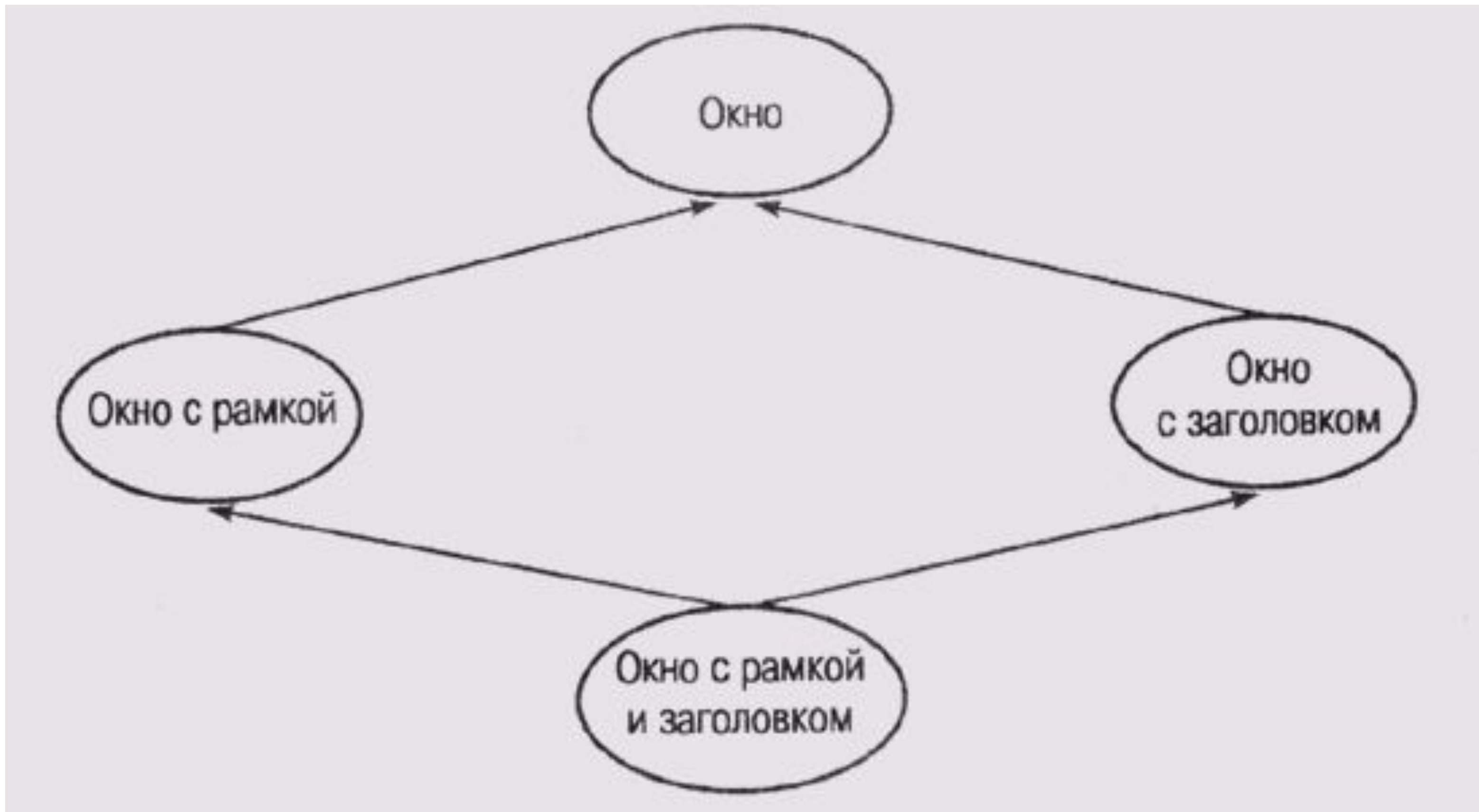


Рис.5 Гетерархическая система классов окон

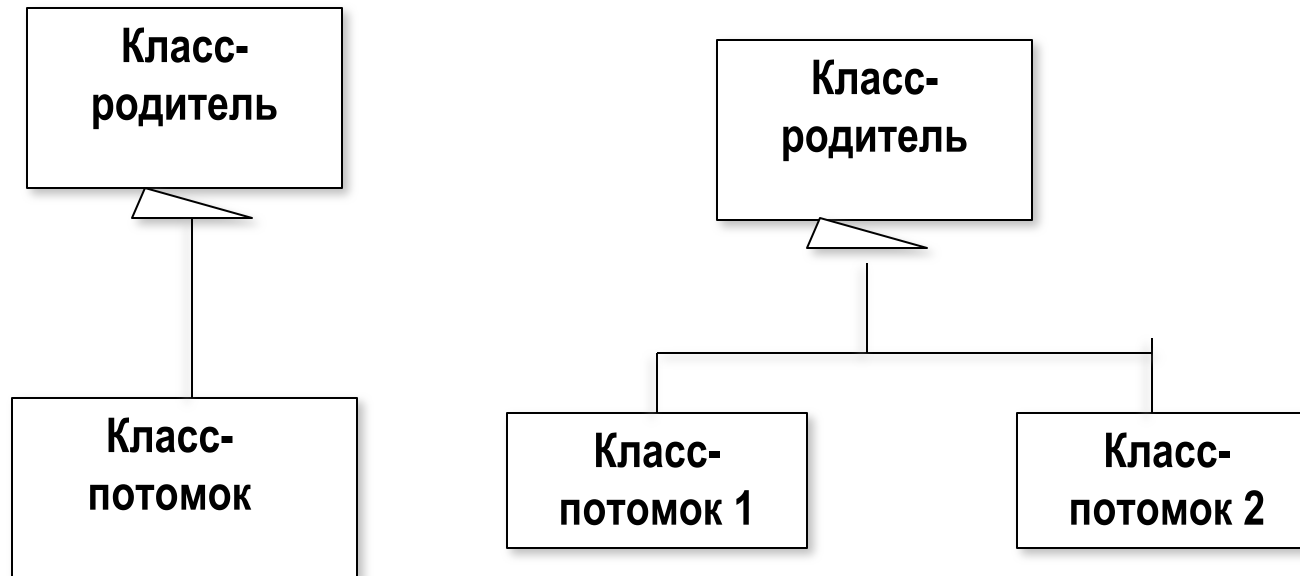


Рис. 6 Примеры иерархий классов

Отношение обобщения обозначается сплошной линией с треугольной стрелкой на конце. Стрелка указывает на более общий класс (класс-предок или суперкласс), а ее отсутствие - на более специальный класс (класс-потомок или подкласс).

Использование наследования способствует уменьшению количества кода, созданного для описания схожих сущностей, а также способствует написанию более эффективного и гибкого кода.

Методы построения классов

Одним из наиболее значимых достоинств ООП является то, что большинство классов для реализации объектов не приходится разрабатывать «с нуля». Обычно классы строят на базе уже существующих, используя механизмы, реализующие определенное отношение существующего и строящего классов между собой: наследование, композицию, агрегацию и полиморфное наследование.

Наследованием или обобщением называют отношение между классами, при котором один класс строится на базе второго посредством добавления полей и определения новых методов. При этом исходный класс, на базе которого выполняется построение, называют *родительским,* или базовым, или супертипом, а строящийся класс - *потомком*, или производным классом, или подтипом.

При наследовании поля и методы родительского класса повторно не определяют, специальный механизм наследования позволяет использовать эти компоненты класса, особо этого не оговаривая. Наследование свойств в иерархии существенно упрощает работу программиста.

В настоящее время созданы библиотеки наиболее часто встречающихся классов, которые можно использовать вновь и вновь, строя на их основе классы для решения различных задач.

Отношения между различными классами проекта принято иллюстрировать диаграммой отношений классов, или просто диаграммой классов.

Если на диаграмме классов показано только отношение наследования, то такую диаграмму называют иерархией классов.

Инкапсуляция (encapsulation) - это сокрытие реализации класса и отделение его внутреннего представления от внешнего (интерфейса). При использовании объектно-ориентированного подхода не принято применять прямой доступ к свойствам какого-либо класса из методов других классов. Для доступа к свойствам класса принято задействовать специальные методы этого класса для получения и изменения его свойств. Несмотря на непривычность слова это просто связывание полей и методов в одну структуру (складывание их в одну "капсулу").

Инкапсуляция позволяет в максимальной степени изолировать объект от внешнего окружения. Она существенно повышает надёжность разрабатываемых программ, т.к. локализованные в объекте алгоритмы обмениваются с программой сравнительно небольшими объёмами данных, причём количество и тип этих данных обычно тщательно контролируется.

В результате замена или модификация алгоритмов и данных, инкапсулированных в объект, как правило, не влечёт за собой плохо прослеживаемых последствий для программы в целом (в целях повышения защищённости программ в ООП почти не используются глобальные переменные).

Полиморфизм

Этот принцип неразрывно связан с наследованием и гласит, что каждый класс наследник может обладать не только свойствами, унаследованными от предка, но и своими собственными. В частности, свойства предка могут быть *перекрыты* наследником - на место свойств предка могут быть подставлены свойства наследника.

Существование принципа полиморфизма является естественным следствием существования принципа наследования: наследование без изменения набора свойств не имеет смысла. Кроме того, без полиморфизма невозможно реализовать объединение различных объектов (классов) по некоторому набору свойств (невозможно абстрагироваться от части свойств объектов), а без этого теряется весь смысл подхода.

Полиморфизм – это свойство родственных объектов (т.е. объектов, имеющих одного общего родителя) решать схожие по смыслу проблемы разными способами. В рамках ООП поведенческие свойства объекта определяются набором входящих в него методов. Изменяя алгоритм того или иного метода в потомках объекта, программист может придавать этим потомкам отсутствующие у родителя специфические свойства. Для изменения метода необходимо перекрыть его в потомке, т.е. объявить в потомке одноимённый метод и реализовать в нём нужные действия.

В результате в объекте-родителе и объекте-потомке будут действовать два одноимённых метода, имеющие разную алгоритмическую основу и, следовательно, придающие объектам разные свойства. Это и называется полиморфизмом объектов

Создание объектов

В Турбо Паскале для создания объектов используется три зарезервированных слова: object, constructor, destructor и три стандартные директивы: private, public и virtual.

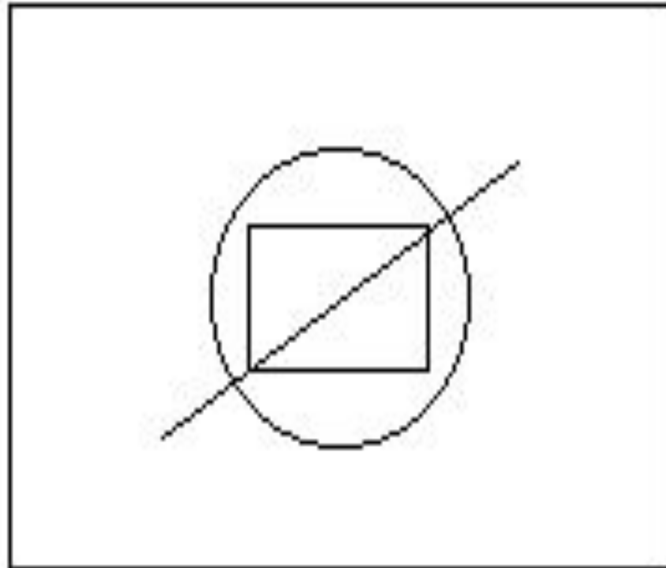
Зарезервированное слово object используется для описания объекта. Описание объекта должно помещаться в разделе описания типов:

```
type
    MyObject = object
    {поля объекта}
     {методы объекта}
     end;
```

Если объект порождается от какого-либо родителя, имя родителя указывается в круглых скобках сразу за словом object:

```
type
   MyDescendantObject = object (MyObject)
 ..........................................................
  ..........................................................
    end;
```

Пример. Требуется разработать программу, которая создаёт на экране ряд графических изображений (точку, окружность, линию, квадрат) и может перемещать эти изображения по экрану. Вид создаваемого программой экрана показан ниже.



Для данной учебной задачи создадим объект-родитель TGraphObject, в рамках которого инкапсулированы поля и методы, общие для всех остальных объектов:

```
type
    TGraphObj = object
     Private  {поля объекта будут скрыты от             пользователя}
            X,Y: Integer;  {координаты точки}
            Color: Word;   {цвет фигуры}
     Public
             Constructor Init (aX, aY: Integer; aColor: Word);
             {создаёт экземпляр объекта}
              Procedure Draw (aColor: Word); Virtual;
              {вычерчивает объект заданным цветом aColor}
              Procedure Show;
              {показывает объект – вычерчивает его цветом Color}
              Procedure Hide;
              {прячет объект – вычерчивает его цветом фона}
              Procedure MoveTo (dX, dY: Integer);
              {перемещает объект в точку с координатами X+dX и Y+dY}
     end;     {конец описания объекта TGraphObj}
```

Чтобы описать все свойства объекта, необходимо раскрыть содержимое объектных методов, т.е. описать соответствующие процедуры и функции. Описание методов производится обычным для Паскаля способом в любом месте раздела описаний, но после описания объекта. Например:

```
type
    TGraphObj = object
    ................................
    end;
Constructor TGraphObj.Init;
begin
      X:= aX;
      Y:= aY;
      Color:= aColor
end;
Procedure TGraphObj.Draw;
begin
{Эта процедура в родительском объекте ничего не делает, поэтому экземпляры TGraphObj не способны отображать себя на экране. Чтобы потомки объекта TGraphObj были способны отображать себя, они должны перекрывать этот метод}
end;
```

```
Procedure TGraphObj.Show;
begin
    Draw (Color)
end;
Procedure TGraphObj.Hide;
begin
    Draw (GetBkColor)
end;
Procedure TGraphObj.MoveTo;
begin
    Hide;
    X:= X+dX;
    Y:= Y+dY;
end;
```

Создадим простейшего потомка от TGraphObj — объект TPoint, с помощью которого будет визуализироваться и перемещаться точка. Все основные действия, необходимые для этого, уже есть в объекте TGraphObj, поэтому в объекте TPoint перекрывается единственный метод — Draw:

```
type
    TPoint = object (TGraphObj)
        Procedure Draw (aColor); Virtual;
    end;
Procedure TPoint.Draw;
begin
      PutPixel (X, Y, Color) {показываем цветом Color
пиксель с    ординатами X и Y}
end;
```

В новом объекте TPoint можно использовать любые методы объекта-родителя TGraphObj. Например, вызвать метод MoveTo, чтобы переместить изображение точки на новое место. В этом случае родительский метод TGraphObj.MoveTo будет обращаться к методу TPoint.Draw, чтобы спрятать и затем показать изображение точки.

Директивы private и public

С помощью директивы *private* при описании объекта можно ограничить область видимости полей и методов. Поля и методы объекта, описанные в секции private, будут доступны только в пределах той программной единицы (программы или модуля), где описан соответствующий объект. Такие поля и методы принято называть приватными.

Директива же *public* просто отменяет действие директивы *private*. То есть все последующие поля и методы будут общедоступными.

Если объектный тип создается "на базе" другого существующего объекта, то имя родительского типа должно быть указано в скобках после слова object при описании потомка:

Type <Потомок> = object(<Родитель>)
<Добавленные поля>
<Добавленные и переопределенные методы>
end;

Как уже говорилось выше, такие объекты автоматически наследуют от родителя его поля и методы. Поля могут быть добавлены (*но не переопределены*), а методы переопределены и добавлены.

Объекты, динамическая память и деструкторы

Экземпляры объектов можно размещать в динамической памяти (куче). Работа с кучей применительно к объектам почти не отличается от работы с кучей применительно к обычным данным. Точно так же необходимо объявить ссылку на объект (переменную ссылочного типа), выделить для нее память в куче, а в конце работы эту память освободить.

Разместим, например, в куче экземпляр объекта TCircle:

Type PCircle = ^TCircle;

Var Circ1 : PCircle; ...

Begin Circ1 := New(PCircle); или New(Circ1);

Для работы с экземпляром объекта, расположенного в куче, необходимо проводить (как обычно) операцию разыменовывания указателя, например:

```
Circ1^.Init(150, 200, 40, Blue);
```

Существует расширенный синтаксис процедуры (и функции) New для выделения памяти экземпляру объекта с одновременным вызовом метода-конструктора:

```
Circ1 := New(PCircle,Init(150,200,40,Blue));
```

или

```
New(Circ1,Init(150,200,40,Blue));
```

При работе с динамическими объектами надо очень серьезно подходить к освобождению памяти. Ведь и сам объект в качестве своих полей может содержать указатели и ссылки. Память, занятая этими динамическими данными, должна освобождаться до вызова процедуры Dispose для самого объекта. То есть если память, занятая полями, освобождается в методе Done, то этот метод должен быть вызван до удаления экземпляра объекта из кучи. То, что экземпляр объекта-родителя и экземпляр объекта-потомка являются совместимыми по типу "сверху вниз" (значение экземпляра объекта-потомка может быть присвоено экземпляру объекта-предка с потерей "лишних данных", но не наоборот), при работе с кучей также может привести к неверному освобождению памяти (может быть освобождена "чужая" память).

В связи с этим синтаксис процедуры Dispose тоже расширен и позволяет вызывать деструктор объекта до его удаления из кучи:

```
Dispose(Circ1,Done);
```

При таком написании, сначала вызывается деструктор, а затем высвобождается память.

Кстати, назначение деструктора как разновидности процедуры - следить за тем, чтобы высвобождалась соответствующая память.

То есть *деструктор* - это, как и конструктор, специальный вид метода, который по-другому еще называют "сборщиком мусора".

Вопросы?