Полиморфизм. Статические, виртуальные и динамические методы. Абстрактные классы. Показать реализацию принципа на предложенном примере.

Виртуальные и динамические методы. Полиморфизм.

По умолчанию методы объектов являются **статическими** — их адреса определяются на этапе компиляции и в программе не изменяются (т.е. вызовом данного имени всегда будет исполняться один и тот же программный код). Статические методы вызываются быстрее всего.

Принципиально от статических отличаются **виртуальные** и **динамические** методы. Они объявляются в разделе описания класса добавлением директивы `virtual` или `dynamic` соответственно. Адреса этих методов определяются во время выполнения программы по специальным таблицам адресов.

Рассмотрим пример. Пусть описывается некоторый класс TData для хранения значения обобщенного типа и три его потомка для хранения строк, целых и вещественных чисел:

```
type

      TData = class
        function Info:string;virtual;abstract;
      end;

      TStringData = class (TData)
        Data:string;
        function Info:string;override;
      end;

      TIntegerData = class (TData)
        Data:integer;
        function Info:string;override;
      end;

      TRealData = class (TData)
        Data:real;
        function Info:string;override;
      end;

function TStringData.Info:string;
  begin
        Info:=Data;
  end;

function TIntegerData.Info:string;
  begin
        Info:=IntToStr(Data);
  end;

function TRealData.Info:string;
  begin
        Info:=FloatToStrF(Data,ffFixed,8,4);
  end;
```

В этом примере классы потомки могут только хранить значения соответствующего типа и выдавать это значение в виде строки. В классе предке нет поля для хранения значения, но описан абстрактный (`abstract`), т.е. не требующий реализации в виде программного кода, метод, одноименный с методами потомков. Если при реализации класса описать и использовать следующий массив:

```
var A : array[1..3].TData;
        i : integer;

procedure ShowData(Adata:TData);
  begin
       Form1.Memo1.Lines.Add(AData.Info);
  end;

begin
  A[1] := TStringData.Create;
  A[2] := TIntegerData.Create;
  A[3] := TRealData.Create;
...
for i:=1 to 3 do ShowData(A[i])
...
end;
```

при каждом вызове ShowData(A[i]) будет вызываться не абстрактный метод `Info` класса TData, соответствующего типу `A[i]` по описанию, а тот метод, который соответствует типу `A[i]` по его определению в тексте программы. Т.е. одной и той же программной строкой вызываются *разные* методы, в зависимости от фактического типа объекта. Данный подход реализует третий из основных принципов объектно-ориентированного программирования - принцип **полиморфизма**. В рамках этого принципа при описании однородных классов вначале описывается абстрактный класс предок, который содержит только интерфейс, а не реализацию одноименных, но различных по тексту методов классов потомков. В потомках же происходит детализация этих методов. Тогда, объявляя объект абстрактного класса-предка, можно вызывать для него тот или иной метод потомка, в зависимости от способа создания объекта (метод Create какого класса был вызван)[9].

Допустим, вы имеете дело с некоторой совокупностью явлений или процессов. Чтобы смоделировать их средствами ООП, нужно выделить их самые общие, типовые черты. Те из них, которые не изменяют своего содержания, должны быть реализованы в виде статических методов. Те же, которые изменяются при переходе от общего к частному, лучше облечь в форму виртуальных методов. Основные, "родовые" черты (методы) нужно описать в классе-предке и затем перекрывать их в классах-потомках. В нашем примере программисту, пишущему процедуру вроде ShowData, важно лишь, что любой объект, переданный в нее, является потомком TData и он умеет сообщить о значении своих данных (выполнив метод `Info`). Если, к примеру, такую процедуру скомпилировать и поместить в динамическую библиотеку, то эту библиотеку можно будет раз и навсегда использовать без изменений, хотя будут появляться и новые, неизвестные в момент ее создания классы-потомки TData!

Наглядный пример использования полиморфизма дает среда Delphi. В ней имеется класс TComponent, на уровне которого сосредоточены определенные "правила" взаимодействия

[9] Здесь необходимо учесть два момента. Во первых, соответствующие варьируемые методы должны быть объявлены как виртуальные или динамические. Во вторых, объекту абстрактного класса недоступны поля потомков (т.е. запись `A[i].Data` в нашем случае была бы ошибочной).

компонентов со средой разработки и с другими компонентами. Следуя этим правилам, можно порождать от TComponent свои компоненты, настраивая Delphi на решение специальных задач.

Теперь — подросшее о виртуальных и динамических методах. Если задуматься над рассмотренным выше примером, становится ясно, что у компилятора нет возможности определить класс объекта, фактически переданного в процедуру showData. Нужен механизм, позволяющий определить это прямо во время выполнения — это называется поздним связыванием (late binding). Естественно, такой механизм должен быть связан с передаваемым объектом. В качестве такого механизма служат таблица виртуальных методов (Virtual Method Table, VMT) и таблица динамических методов (Dynamic Method Table, DMT).

Разница между виртуальными и динамическими методами заключается в особенности поиска адреса. Когда компилятор встречает обращение к виртуальному методу, он подставляет вместо прямого вызова по конкретному адресу код, который обращается к VMT и извлекает оттуда нужный адрес. Такая таблица есть для каждого класса (объектного типа). В ней хранятся адреса всех виртуальных методов класса, независимо от того, унаследованы ли они от предка или перекрыты в данном классе. Отсюда и достоинства и недостатки виртуальных методов: они вызываются сравнительно быстро, однако для хранения указателей на них в таблице VMT требуется большое количество памяти.

Динамические методы вызываются медленнее, но позволяют более экономно расходовать память. Каждому динамическому методу системой присваивается уникальный индекс. В таблице динамических методов класса хранятся индексы и адреса только тех динамических методов, которые описаны в данном классе. При вызове динамического метода происходит поиск в этой таблице; в случае неудачи просматриваются таблицы DMT всех классов-предков в порядке иерархии и, наконец, TObject, где имеется стандартный обработчик вызова динамических методов. Экономия памяти налицо. Те, для кого это не очевидно или недостаточно, найдут подробности в разделе данной главы "Как у строен объект изнутри".

Т.о., различие виртуальных и динамических методов состоит только в способе хранения из адресов. Адрес виртуального метода дальнего предка можно быстро взять из таблицы VMT данного класса, а для поиска адреса динамического метода дальнего предка придется по очереди перебирать всю иерархическую цепочку данного класса, что гораздо медленнее, но экономит память[10].

Директива override указывает что перекрывается динамический или виртуальный метод класса-предка. Без этой директивы перекрытие перекрывается динамических или виртуальных методов невозможно.

[10] В наше время сверхбыстрых компьютеров с огромной памятью об этом можно не беспокоиться.