Глава 13. Объектно-ориентированное программирование в языке Perl

Учебник по Perl

Содержание [скрыть]

Эта глава не предназначена для того, чтобы изучать по ней основы объектно-ориентированного программирования (ООП). Мы лишь хотим дать представление о том, как основные идеи ООП реализованы в языке Perl. Начнем с краткого обзора этих идей. В основе ООП лежат понятия *класса* и *объекта.* Эти понятия тесно связаны друг с другом.

Класс представляет собой сочетание структуры данных и тех действий, которые можно выполнить над этими данными. Данные называют *свойствами,* а действия — *методами.* Совмещение в классе структуры данных и действий над ними называют *инкапсуляцией.*

Объект является экземпляром класса. Свойства объекта обусловлены его принадлежностью к определенному классу. Понятия "объект" и "класс" отражают два различных вида иерархий, которые можно обнаружить в любой достаточно сложной системе. Рассмотрим пример.

Персональный компьютер является сложной системой. Жесткий диск — составная часть этой системы. Другими ее частями являются центральный процессор, память и т. д. Можно сказать, что жесткий диск — это часть *структурной* иерархии под названием "персональный компьютер". С другой стороны, жесткий диск является абстракцией, обобщением свойств, присущих всем жестким дискам, и этим он отличается, например, от *гибкого* диска, рассматриваемого как абстракция, обобщающая свойства всех гибких дисков. В *типовой* иерархии жесткие диски Quantum — это особый тип жестких дисков, жесткие диски Quantum Fireball EL — особый тип жестких дисков Quantum, жесткие диски Quantum Fireball EL объемом 5,1 Гбайт — особый тип жестких дисков Quantum Fireball EL и т. д.

В данной аналогии абстрактный жесткий диск является *базовым* классом, жесткие диски фирмы Quantum образуют *подкласс* класса жестких дисков, жесткие диски Quantum Fireball EL – подкласс класса жестких дисков фирмы Quantum и т. д. Подкласс называют также *производным* классом или классом-потомком. Для него класс, расположенный выше в иерархии, является базовым, *подклассом* или родительским классом. В нашем примере *объект* (конкретный жесткий диск) является составной частью *структуры* под названием "персональный компьютер", и обладает своими свойствами в силу принадлежности к определенному *типу* дисков Quantum Fireball EL 5,1 Гбайт, отличающемуся по своим характеристикам от других типов жестких дисков.

Действие, которое можно выполнить над данными, определяет *метод.* Он представляет собой подпрограмму. Различают *методы класса* и *методы объекта.* Последние могут применяться к любому объекту класса. Методы класса, называемые также *статическими* методами, не зависят от отдельного экземпляра, они применяются к целому классу как к отдельному объекту.

Для создания объектов применяются специальные статические методы, называемые *конструкторами.*

Для корректного удаления объектов используются специальные методы, называемые *деструкторами.*

Термин *наследование* в ООП обозначает способность классов наследовать свойства и методы у своих родительских классов.

Термин *полиморфизм* в ООП обозначает свойство класса-потомка переопределять методы родительского класса.

Инкапсуляция, наследование, полиморфизм – базовые понятия, лежащие в основе объектно-ориентированного программирования. Объектно-ориентированный подход возник в результате непрекращающихся попыток человека преодолеть сложность окружающего мира, проявляющуюся в любых областях деятельности. Он предлагает более естественный способ разделения сложной задачи на ряд более простых, используя понятие объекта, сочетающего данные и действия над ними. Он также позволяет экономить усилия, связанные с написанием программного кода, так как однажды созданные методы не нужно переписывать – они просто наследуются производными классами. Идея повторного использования кода в своем развитии проходит ряд этапов: подпрограмма, библиотека, модуль, класс.

В этой главе мы рассмотрим, каким образом основные концепции объектно-ориентированного программирования реализованы в языке Perl.

Классы и объекты

В языке Perl нет специального синтаксиса для описания классов, объектов или методов. Для их реализации используются уже знакомые нам синтаксические конструкции. Класс в Perl представляет собой пакет, объект является ссылкой, а метод – обычной подпрограммой.

В качестве пакета каждый класс имеет собственное изолированное пространство имен и таблицу символов, реализованную в виде хеш-массива. К переменным класса можно обращаться, используя их квалифицированные имена, содержащие в качестве префикса имя класса, за которым следуют два двоеточия, например, $CLASSNAME: :var. Для того чтобы пакет стал классом, в нем нужно определить специальную подпрограмму – конструктор, которая используется для создания отдельных экземпляров класса – объек- \ тов *(см. раздел 13.2.1).* \

Объект в Perl представляет собой просто ссылку, но^не любую, а связанную с определенным классом. Тип ссылки можно определить при помощи функции ref EXPR, которая, рассматривая свой аргумент как ссылку, возвращает символическое обозначение ее типа. Для встроенных типов Perl используются следующие символические обозначения:

- REF – ссылка на ссылку;
- SCALAR – ссылка на скаляр;
- ARRAY – ссылка на массив;
- HASH – ссылка на ассоциативный массив;
- CODE – ссылка на подпрограмму;
- GLOB – ссылка на переменную типа typegiob.

Для ссылки-объекта функция ref () возвращает имя класса, к которому этот объект принадлежит. Обычная ссылка становится объектом после ее "посвящения" в члены класса при помощи функции

```
bless REF [, CLASSNAME]
```

Слово "bless" в английском языке имеет значение "освящать, благословлять". В данном контексте его можно перевести как "посвящать" или "санкционировать". Функция bless REF санкционирует принадлежность субъекта ссылки REF к классу CLASSNAME. Она возвращает ссылку на этот субъект, но уже другого типа – CLASSNAME (напомним, что в главе 9 субъектом ссылки мы условились называть то, на что она указывает, т. е. собственно структуру данных некоторого типа). Она связывает обычную ссылку с именем класса. Если имя класса не задано, то используется имя текущего класса. После выполнения функции bless ок созданному ей объекту можно обращаться, используя его квалифицированное имя $CLASSNAME: :REF. Сказанное иллюстрируется следующим примером.

```
$h = { }; ·
print("тип переменной \$h - ". ref($h), "\n");
bless($h, "MyClass");
print("тип переменной \$h - ". ref($h), "\n");
```

В результате будет выведен тип переменной $ref до и после вызова функции bless ():

```
тип переменной $h – HASH тип переменной $h – MyClass
```

Наследование в Perl отличается от наследования в других объектно-ориентированных языках программирования тем, что наследуются только методы. Наследование данных реализуется программистом самостоятельно. Наследование методов реализовано следующим образом. С каждым пакетом ассоциирован свой специальный массив @ISA, в котором хранится список базовых классов данного пакета. Таким образом, подкласс располагает информацией о своих базовых классах. Если внутри текущего класса встречается обращение к методу, не определенному в самом классе, то интерпретатор в поисках отсутствующего метода просматривает классы в том порядке, в котором они встречаются в массиве @ISA. Затем просматривается предопределенный класс UNIVERSAL. В нем изначально нет никаких явных определений, но автоматически содержатся некоторые общие методы, которые неявно наследуются всеми классами. В этом смысле класс UNIVERSAL можно считать базовым классом всех остальных классов.

Если метод не найден ни в одном из просмотренных классов, они вновь просматриваются в том же порядке в поисках подпрограммы AUTOLOAD *(см. раздел 12.1.3).* Если таковая обнаружена, она выполняется вместо вызванного несуществующего метода с теми же параметрами. Квалифицированное имя несуществующего метода при этом доступно через переменную $AUTOLOAD.

Методы

Методы в Perl являются обычными подпрограммами. Начнем их изучение с методов, которые обязательно должны быть определены в каждом классе. Такими методами являются конструкторы. Знакомство с ними позволит лучше понять, каким способом в языке Perl представляются объекты.

Конструкторы

Конструктор – это просто подпрограмма, возвращающая ссылку. Обычно (но не обязательно) конструктор имеет имя new. Выше мы сказали, что объект является ссылкой. Конструктор, создающий объект, то есть ссылку, тоже возвращает ссылку. О каких ссылках идет речь, на что они указывают? Рассмотрим простой пример.

```
package MyClass; sub new {
my $class = shift;
my $self = {} ;
bless $self, $class;
return $self; -""'
} ^
```

В операторе my $seif = {} создается ссылка на[7] анонимный хеш-массив, первоначально пустой, которая сохраняется в локальной переменной $seif. Функция bless о "сообщает" субъекту ссылки sseif, то есть анонимному хеш-массиву, что он отныне является объектом класса MyClass, и возвращает ссылку на этот объект. Затем ссылка на новый объект класса MyClass возвращается в качестве значения конструктора new (). Обратите внимание на следующие обстоятельства.

Во-первых, объект класса MyClass представлен анонимным хеш-массивом. Это обычный, хотя и не обязательный, способ представления объекта. Преимущество использования для этой цели хеш-массива заключается в том, что он может содержать произвольное число элементов, к которым можно обращаться по произвольно заданному ключу. Таким образом, все данные объекта сохраняются в указанном хеш-массиве. В некоторых случаях для представления объекта может использоваться другой тип данных, например массив, скаляр.

Во-вторых, обязательным для конструктора является обращение к функции bless o. Внутри подпрограммы-конструктора функцию bless o следует применять с двумя аргументами, явно указывая имя класса. В этом случае конструктор может быть наследован классом-потомком. Конструктор определяет, объект какого именно класса он создает, используя значение своего первого аргумента. Первым аргументом конструктора должно быть имя класса.

В-третьих, конструктору, создающему объект, в качестве первого аргумента передается имя класса этого объекта.

```
f Модуль класса Staff.pm: package Staff; require Exporter; 8ISA = qw(Exporter); SEXPORT =
qw(new); sub new {
my ($class,@items) = shift;
my $self = {};
bless $self, $class;
return $self; }
# Основная программа:
#!/usr/bin/perl use Staff;
$someone=new(Staff) ; ${$someone}{"имя"}="Александр"; ${$someone}{"фамилия"}="Александров";
${$someone}{"возраст"}="37"; for $i (sort keys %{$someone}) { print "$i=>$$someone{$i}\n"; }
```

В данном примере класс staff служит для представления анкетных данных. В качестве внутренней структуры для представления объекта наилучшим образом подходит хеш-массив, так как в него при необходимости можно добавлять новые элементы с произвольно заданными ключами, например, "имя", "фамилия", "образование" и т. д. Класс оформлен в виде отдельного модуля, способного управлять экспортом своих методов. Чтобы конструктор new () можно было вызвать в основной программе, он включен в файл экспорта @EXPORT. В результате вызова конструктора возвращается ссылка на объект класса. Значения элементов хеш-массива выводятся:

```
возраст => 37
имя => Александр
фамилия => Александров
```

Методы класса и методы объекта

Различают *методы класса* и *методы объекта,* которые называют также *статическими* и *виртуальными,* соответственно. Первым аргументом метода должна быть ссылка, представляющая объект, или имя пакета. То, каким образом каждый метод обрабатывает свой первый аргумент, определяет, является ли он методом класса или методом объекта.

Статические методы применяются к целому классу, а не к отдельным его объектам. В качестве первого аргумента статическому методу передается имя класса. Типичным примером статических методов являются конструкторы. В Perl методы выполняются в пространстве имен того пакета, в котором они были определены, а не в пространстве имен пакета, в котором они вызваны. Поэтому статические методы часто свой первый аргумент игнорируют, так как и без него известно, к какому пакету метод принадлежит. Последнее не относится к конструкторам, передающим свой первый аргумент функции Wesz () в качестве имени класса.

Методы объекта применяются к отдельному объекту. Их первым аргументом должна быть ссылка, указывающая на объект, к которому применяется данный метод. Методы объекта могут выполнять разные действия, но наиболее типичными являются действия, связанные с отображением и изменением данных, инкапсулированных в объекте. В примере 13.2 мы присвоили значения, а затем их распечатали, обращаясь к данным объекта напрямую. Таким способом мы просто продемонстрировали, что конструктор действительно возвращает ссылку, представляющую объект. В действительности это плохой способ, и применять его не рекомендуется. В некоторых объектно-ориентированных языках программирования прямой доступ к данным объекта вообще невозможен. Объект следует рассматривать как "черный ящик", содержимое которого можно получить или изменить, только используя методы объекта. Такое ограничение помогает сохранить целостность и скрыть некоторые внутренние данные объекта. Отредактируем текст примера 13.2, дополнив его методами, которые используются для изменения и просмотра данных.

```
# Модуль класса Staff.pm:
package Staff;
require Exporter;
@ISA = qw(Exporter);
8EXPORT = qw(new showdata setdata);
sub new {
my ($class, $data) = @_;
my $self = $data;
bless $self, $class;
return $self; } sub showdata {
my $self = shift;
my @keys = @_ ? @_ : sort keys %$self;
foreach $key (@keys) {
print "\t$key => $self->{$key}\n";
```

```
}
return $self;
}
sub setdata {
my ($self,$data) = @_; for $i (keys %$data) {
$self->{$i)=$data->{$i);
}
return $self;
}
```

В данном примере по сравнению с предыдущим изменен конструктор new (). Теперь второй параметр, представленный локальной переменной $data, содержит ссылку. Эту ссылку функция bless () свяжет с классом staff, превратив в его объект. Таким образом, при помощи этого параметра можно управлять типом внутренней структуры данных, которая и представляет объект. Это может быть ссылка на хеш-массив, массив, скаляр и т. д. Параметры, передаваемые конструктору, называют *переменными объекта.* Они используются для того, чтобы установить начальные значения данных каждого вновь создаваемого объекта.

Если обратиться к основной программе:

```
#!/usr/bin/perl
use Staff;
$someone=new(Staff, ("имя"=>"","фамилия"=>""});
setdata($someone,{"имя"=>"Максим","фамилия"=>"Исаев",
"возраст"=>42,"занятия спортом"=>"теннис"}); showdata($someone);
```

то будут выведены следующие данные:

```
возраст => 42
занятия спортом => теннис
имя => Максим
фамилия => Исаев
```

В разных ситуациях один и тот же метод может выступать как метод класса или как метод объекта. Для этого он должен "уметь" определить тип своего первого аргумента: если аргумент является ссылкой, то метод действует как метод объекта, если именем пакета, то есть строкой, то как метод класса. Подобную информацию можно получить при помощи функции ref (). Она возвращает значение ЛОЖЬ (пустая строка), если ее аргумент не является ссылкой, то есть объектом. В противном случае функция ref о возвращает имя пакета, принадлежность к которому была для данного объекта санкционирована функцией bless ().

Вызов метода

Существуют две синтаксические формы вызова как методов класса, так и методов объекта.

Первая форма имеет вид:

```
method class_or_object, parameters
```

например,

```
$somebody = new Staff, {"имя"=>"Анна"}; # метод класса
showdata $somebody, "имя","фамилия"; # метод объекта
showdata {"имя"=>"Мария","возраст"=>18}; # метод объекта
showdata new Staff "возраст"; # метод объекта
showdata setdata hew Staff, {"имя"=>"Глеб"}, "имя"; # метод объекта
```

Данная форма представляет собой обычный вызов функции, который может быть вложенным в другой вызов. Первым аргументом функции является ссылка (для методов объекта) или/им^Гпакета (для методов класса).

В приведенном примере первая строка содержит вызов конструктора new, в котором первым (и единственным) аргументом является имя пакета.

Вторая строка содержит вызов метода объекта, в котором первым аргументом является объект-ссылка.

В третьей строке первый аргумент задается при помощи блока {}, возвращающего ссылку на анонимный хеш-массив. Данный хеш-массив не будет объектом, так как он не объявлен объектом класса staff при помощи функции bless o, но синтаксически такая конструкция возможна.

В четвертой строке метод объекта вызывается с двумя аргументами. Первым аргументом является ссылка, возвращаемая конструктором new (), вторым – строка "возраст".

В пятой строке конструктор new создает объект, который передается в качестве первого аргумента методу setdata. Вторым аргументом метода setdata является ссылка на анонимный хеш-массив {"имя"=>"Глеб"). Метод showdata в качестве первого аргумента использует ссылку, возвращаемую методом setdata, а в качестве второго аргумента – строку "имя".

Вторая форма обращения к методу имеет вид

```
class_or_obj ect ->method(parameters)
```

Например, предыдущие вызовы могут быть записаны также в виде:

```
$somebody = Staff->new(("имя"=>"Анна"});
$somebody->showdata("имя","фамилия");
```

```
new Staff->showdata("возраст");
new Staff->setdata({"имя"=>"Глеб"})->showdata("имя");
```

Вторая форма требует обязательного заключения аргументов в скобки.

Замечание

Как видим, обе формы записи могут быть достаточно сложными. В разных случаях любая из них может оказаться предпочтительной в смысле читаемости текста программы. Если нет серьезных оснований против, то рекомендуем использовать вторую форму, исходя из следующих соображений.

При использовании первой формы на том месте, где должен стоять объект или имя класса, синтаксис позволяет использовать либо имя класса, либо скалярную переменную, содержащую ссылку, либо блок {...}, возвращающий значение ссылки. Вторая форма для Представления объекта позволяет использовать более сложные конструкции, например, элемент хеш-массива:

```
$obj->{keyname}->method().
```

При употреблении первой формы могут возникнуть трудноопределимые ошибки. Например, если происходит обращение к методу в области Видимости одноименной функции, то при использовании первой формы вызова компилятор может вместо метода вызвать функцию. Вторая форма подобную ошибку исключает.

Для того чтобы вызвать метод определенного класса, следует перед именем метода указать имя пакета, как при вызове обычной подпрограммы. Например, чтобы вызвать метод, определенный в пакете staff, следует использовать запись вида:

```
$someone = new Staff;
Staff::showdata($someone, "имя");
```

В данном случае просто вызывается метод showdata из пакета staff. Ему передается в качестве аргумента объект $ some one и прочие аргументы. Если вызвать метод следующим образом:

```
$someone=new Staff; $someone->Staff::showdata("имя");
```

то это будет означать, что для объекта $ someone следует сначала найти метод showdata, начав поиск с пакета staff, а затем вызвать найденный метод с объектом $ someone в качестве первого аргумента. Напомним, что с каждым пакетом ассоциируется свой массив @ISA, содержащий имена других пакетов, представляющих классы. Если интерпретатор встречает обращение к методу, не определенному в текущем пакете, он ищет этот метод, рекурсивно (то есть включая производные классы) просматривая пакеты, определенные в массиве @ISA текущего пакета. Если подобный вызов

делается внутри пакета, являющегося классом, то для того, чтобы указать в качестве начала поиска базовый класс, не указывая его имя явно, можно использовать имя псевдокласса SUPER:

```
$someone->SUPER::showdata("имя");
```

Такая запись имеет смысл только внутри пакета, являющегося классом.

Деструкторы

В главе 9 было сказано, что для каждого субъекта ссылки поддерживается счетчик ссылок. Область памяти, занимаемая субъектом ссылки, освобождается,, когда значение счетчика ссылок становится равным нулю. Объект, как мы знаем, является просто ссылкой, поэтому с ним происходит то же самое: когда значение счетчика ссылок становится равным нулю, внутренняя структура данных, представляющая объект (обычно хеш-массив), освобождает память. Интерпретатор сам отслеживает значение счетчика ссылок и автоматически удаляет объект. Пользователь может определить собственные действия, завершающие работу объекта, при помощи специального метода – деструктора. Деструктор нужен для того, чтобы корректно завершить жизненный цикл объекта, например, закрыть открытые объектом файлы или просто вывести нужное сообщение. В соответствующее время деструктор будет автоматически вызван интерпретатором.

Деструктор должен быть определен внутри своего класса. Он всегда имеет имя DESTROY, а в качестве единственного аргумента – ссылку на объект, подлежащий удалению. Создавая подпрограмму-деструктор, следует обратить внимание на то, чтобы значение ссылки на удаляемый объект, передаваемое в качестве первого элемента массива параметров $_[0], не изменялось внутри подпрограммы.

Подчеркнем, что создавать деструктор не обязательно. Он лишь предоставляет возможность выполнить некоторые дополнительные завершающие действия. Основная работа по удалению объекта выполняется автоматически. Все объекты-ссылки, содержащиеся внутри удаляемого объекта как его данные, также удаляются автоматически. /

Метод DESTROY не вызывает другие деструкторы автоматически. Рассмотрим следующую ситуацию. Конструктор класса, вызывая конструктор своего базового класса, создает объект базового класса. Затем при помощи функции bless о делает последний объектом собственного класса. Оба класса, текущий и базовый, имеют собственные деструкторы. Поскольку конструктор базового класса, вызванный конструктором текущего класса, создал собственный объект, то при его удалении должен вызываться деструктор базового класса. Но этот объект уже перестал быть объектом базового класса, так как одновременно объект может принадлежать только одному классу. Поэтому при его удалении будет вызван только деструктор текущего класса. При необходимости деструктор текущего класса должен вызвать деструктор своего базового класса самостоятельно.

Обобщающий пример

В заключение рассмотрим небольшой пример, поясняющий некоторые вопросы, рассмотренные в этой главе.

```
#!/usr/bin/perl package Staff; sub new {
my ($class, $data) = @_;
my $self = $data;
bless $self, $class;
return $self; } sub setdata {
my ($self,$data) = @_;
for $i (keys %$data) {
$self->{$i}=$data->{$i};
}
return $self; } sub showdata {
my $self = shift;
my @keys = @_ ? @_ : sort keys %$self;
foreach $key (@keys) {
print "\t$key => $self->{$key}\n";
}
return $self; } sub AUTOLOAD {
print "пакет Staff: отсутствует функция $AUTOLOAD\n"; } sub DESTROY {
print "Удаляется объект класса Staff\n"; }
########################################
package Graduate; @ISA = (Staff); sub new {
my ($class, $data) = @_;
# наследование переменной объекта
my $self = Staff->new($data);
$self->{"образование"}="высшее";
bless $self, $class;
return $self; } sub showdata {
my $self = shift;
return $self if ($self->{"образование"} пе "высшее");
my @keys = sort keys %$self;
foreach $key (@keys) {
print "\t$key => $self->{$key}\n";
}
return $self; } sub DESTROY {
my $self= shift;
$self->SUPER::DESTROY;
print "Удаляется объект класса Graduate\n";
```

```
####################################
package main;
$someone=Graduate->new({ "фамилия" => "Кузнецов", "имя" => "Николай" });
$somebody=Staff->new({"фамилия" => "Петрова", "имя" => "Анна"});
$someone->showdata;
$somebody->Graduate::showdata;
$someone->getdata;
```

Для простоты все классы расположены в одном файле. Если класс занимает отдельный модуль, необходимЬчхрзабхшпъся об управлении экспортом имен при помощи списков @EXPORT и @EXPORT_OK, а также о подключении соответствующих модулей к вызывающей программе *(см. раздел 12.3).*

В данном примере определен пакет main и два класса: staff и Graduate, staff является *базовым* классом Graduate.

Наследование методов задается при помощи массива @ISA, ассоциированного с *производным* классом Graduate. Помимо наследования методов, которое обеспечивает Perl, организовано *наследование переменных, объекта* через вызов в конструкторе класса Graduate конструктора базового класса staff. В результате объект класса Graduate получает при создании *переменную объекта* staff (параметр, переданный конструктору new), которую изменяет, добавляя в соответствующий хеш-массив новый элемент с ключом "образование" и значением "высшее".

Ситуация, когда конструктор производного класса вызывает конструктор базового, также обсуждалась в разделе 13.2.4. В подобном случае при удалении объекта автоматически вызывается только деструктор производного класса, хотя при вызове конструктора последовательно создавались объекты двух классов. В примере деструктор DESTROY класса Graduate вызывает деструктор базового класса. В данном случае это совершенно не обязательно, так как оба деструктора просто выводят сообщения об удалении своих объектов, но ведь это только модель. В иной ситуации такое решение может быть необходимым.

Следующая тема связана с поиском несуществующего метода, к которому происходит обращение внутри класса, и применением в этом случае функции AUTOLOAD. В пакете main вызывается метод getdata. для объекта $someone. $someone **является объектом класса** Graduate **в котором метод getdata не определен. Обратите внимание на форму вызова метода getdata. Если бы он был вызван в виде getdata ($someone), это означало, что вызывается функция getdata из пакета main с параметром $someone. Поскольку в пакете main такая функция не определена, выполнение программы было бы завершено с выдачей сообщения вида:**

```
Undefined subroutine Smain::getdata called at ...
```

Форма обращения $someone->getdata однозначно определяет, что вызывается не функция, а метод объекта. Следовательно, его надо искать сначала в собственном классе Graduate, а затем в классе

staff, который определен в качестве базового для Graduate. В классе staff метод getdata также не определен, но определена функция AUTOLOAD, которая и вызывается вместо этого метода.

Полиморфизм, т. е. переопределение методов базового объекта, показан на примере метода showdata. Класс staff служит для порождения анкетных форм учета персонала. Его подкласс Graduate описывает подмножество таких форм только для лиц с высшим образованием. Конструктор класса Graduate автоматически добавляет в форму запись о наличии высшего образования. Метод showdata, наследованный у базового класса, изменен таким образом, чтобы игнорировать анкеты без такой записи. Поэтому информация об объекте $ somebody, принадлежащем к базовому классу, напечатана не будет.

В результате выполнения примера будут выведены строки, соответствующие действиям, заданным в программе:

```
имя => Николай образование => высшее фамилия => Кузнецов
пакет Staff: отсутствует функция
Graduate::getdata
Удаляется объект класса Staff
Удаляется объект класса Graduate
Удаляется объект класса Staff.
```

Вопросы для самоконтроля

1. Как связаны между собой понятия "объект" и "класс"?
2. Что такое инкапсуляция, наследование, полиморфизм в объектно-ориентированном программировании?
3. Как в языке Perl реализовано понятие "класс"?
4. Что представляет собой объект в языке Perl?
5. Для чего предназначена функция bless () ?
6. Как, по вашему мнению, связаны между собой понятия "класс", "пакет", "модуль"?
7. Как в Perl осуществляется наследование методов? Данных?
8. Какой синтаксис используется для объявления метода?
9. Чем конструкторы в Perl отличаются от других методов?
10. В чем разница между методами класса и методами объекта?
11. Какие формы вызова объекта вы знаете? В чем, по-вашему, заключаются их преимущества и недостатки?
12. Какие имена может иметь подпрограмма-деструктор?
13. Как можно реализовать переопределение метода базового класса (полиморфизм)?
14. Как осуществить наследование переменных объекта?

Упражнение

Создайте класс, объект которого представляет дерево файловой системы для заданного каталога и позволяет вывести это дерево с указанием всех вложенных каталогов и содержащихся в них файлов.