**1)** Модуль в программировании представляет собой функционально законченный фрагмент программы, оформленный в виде отдельного файла с исходным кодом, предназначенный для использования в других программах. Модуль – единица разработки, применения и поставки. Модули позволяют разбивать сложные задачи на более мелкие в соответствии с принципом модульности. Основные концепции модульного программирования:

-каждый модуль имеет единственную точку входа и выхода;

-размер модуля по возможности должен быть минимизирован;

-вся система построена из модулей;

-каждый модуль не зависит от того, как реализованы другие модули.

Разграничение доступа к модулю осуществляется с помощью ключевых слов: public, private, protected. Язык Java позволяет организовать модульность с использованием файла module-info.java. Для этого в файлах двух модулей прописывается exports и requires соответственно. Объект – динамический модуль, у которого может быть много экземпляров. Описание динамического модуля – класс. Изначальное значение слова «инкапсуляция» в программировании — объединение данных и методов работы с этими данными в одной упаковке («капсуле»). В Java в роли упаковки-капсулы выступает класс. Класс содержит в себе и данные (поля класса), и методы для работы с этими данными. Теперь о сокрытии. Все как в реальной жизни: пользователю предоставлен интерфейс (методы). Если ему нужно, чтобы автомобиль в программе выполнил действие, достаточно вызвать нужный метод. Именно в этом заключается суть сокрытия. Все «внутренности» программы скрываются от пользователя. Для него эта информация является лишней, ненужной.

**2)** Исключение – событие и объект с информацией об ошибке.

Исключение создается: Аппаратно – в результате ошибки выполнения оператора. Программно – с помощью оператора создания исключения: throw new SQLException(“Hello”). Обрабатывать исключения можно по месту с помощью try catch. Можно делать и частичную обработку, делая throw new Exception и пробрасывая его на уровень выше. У исключений есть свои классы. Например в Java есть класс Throwable, от которого наследуются Error и Exception. Exception класс позволяет отловить любое проверяемое компилятором исключение. Есть проверяемые и непроверяемые исключения(RuntimeException). Хорошим тоном в программировании является защита ресурсов при возникновении исключения. Примером может служить закрытие подключения к базе данных(или освобождение другого ресурса) в блоке finally. Еще примером является использование оператора try with resources(или using в c#)

**3)** Класс – тип данных, шаблон для создания объектов. Объект – экземпляр класса. Объект – экземпляр класса. Конструктор и деструктор – особые методы. Конструктор принимает непосредственное участие в создании объекта. Для создания объекта(выделения для него памяти) используется ключевое слово new. При этом, вызывается метод конструктор, который и возвращает новый объект. В конструкторе может производится инициализация полей объекта или другие действия. В свою же очередь, в ряде языков программирования с ручным управлением памятью предусмотрены деструкторы. Это методы, которые освободят память и уничтожат объект. На самом деле это всего лишь сигнал системе, что эта память может использоваться под другие нужды.

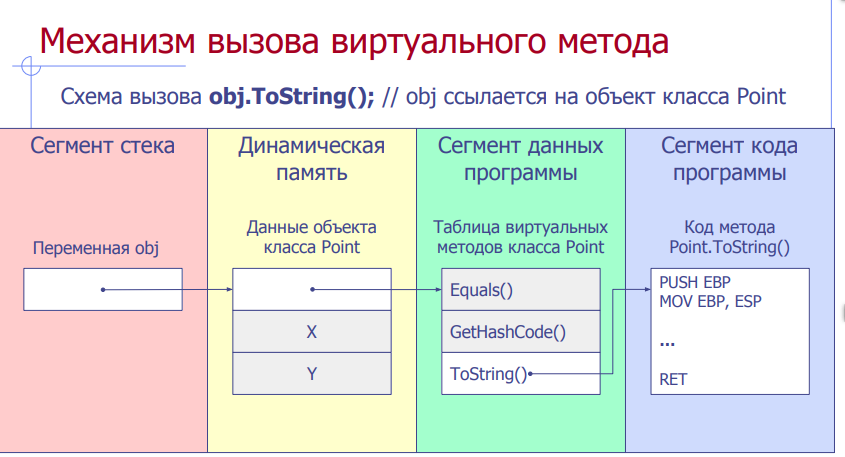
**4)** Свойство – виртуальное поле. В C# есть отдельное понятие:

public bool Active { get { return fActive; } set { SetActive(value); } }

В данном случае объявлено два свойства для одного поля. Они являются своего рода аналогами геттеров и сеттеров в Java. Но особенность в том, что в них можно вкладывать и дополнительную логику при обращении к данному полю класса и получать разные значения. С другой стороны, такую же функциональность можно встроить и в обычный getter и setter в языке Java, хоть это считается не очень хорошим тоном.

**5)** Одним из ключевых аспектов объектно-ориентированного программирования является наследование. С помощью наследования можно расширить функционал уже имеющихся классов за счет добавления нового функционала или изменения старого. При изменении старого функционала происходит переопределение методов класса родителя. При этом класс наследник имеет доступ ко всем членам класса родителя, кроме тех, что помечены как private. И в java, и в c# для всех новообразованных классов имеется суперкласс. В java это Object, в c# - System.Object. В этом классе содержаться базовые методы для всех классов по типу equals, hashCode, toString. Любой класс наследник может быть приведен к классу родителю. То есть ссылке на объект класса родителя можно присвоить ссылку на объект наследник. Контроль типов в java осуществляется с помощью instance of, в c# - это ключевое слово is. Приведение типов традиционное: (Type)object. Но в c# имеется и оператор as, который в отличие от обычного способа не выбрасывает исключения.

**6)** Виртуальный метод – переопределяемый в производных классах метод. Вызов виртуального метода осуществляется в соответствии с фактическим типом объекта, к которому метод применяется. При этом такие методы могут иметь реализацию при желании могут быть переопределены в наследнике. В java такие методы не помечаются как virtual.



В обоих языках есть абстрактные методы. При этом и сам класс в java должен быть помечен абстрактным. При этом классы наследники обязательно должны реализовывать(переопределены) такие методы. А объекты класса родителя запрещены к созданию. В некоторых языках(не в c#) есть поддержка динамических виртуальных методов. Отличие с обычными виртуальными методами заключается в том, что динамические методы вызываются дольше, но имеют более компактный код и экономят память.

Полиморфизм – это способность программы идентично использовать объекты с одинаковым интерфейсом без информации о конкретном типе этого объекта.

Dancer breakDanceDancer = new BreakDankDancer("Алексей", 19);

// восходящее преобразование к базовому типу

Dancer electricBoogieDancer = new ElectricBoogieDancer("Игорь", 20);

// восходящее преобразование к базовому типу

В c# свойство(поля) можно сделать виртуальным. Тогда в производных классах перекрываются(переопределяются) его методы чтения и записи. При перекрытии свойства нельзя изменять атрибуты доступа методов чтения и записи. Запрет на расширение(наследование) в c# происходит с помощью слова sealed для классов и виртуальных методов, в java – это слово final(тоже для классов и методов).

**7)** Делегат – ссылка на метод объекта:

public delegate bool NextLineDelegate();

Теперь можно объявить переменную с таким типом и присвоить ей ссылку на метод какого-то объекта (сигнатура метода должна совпадать с сигнатурой делегата)

NextLineDelegate NextLineDelegate = Form1.NextLine;

Наконец, можно сделать вызов через процедурную переменную: NextLineDelegate();

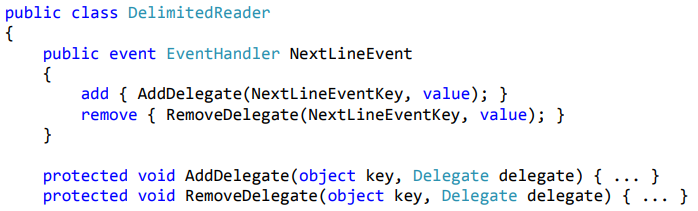
В результате этого оператора у объекта Form1 будет вызван метод bool NextLineNotification(), потому что в делегате хранится как ссылка на объект, так и ссылка на метод. Похожее есть и в Java: лямбда-выражения.

Событие представляет собой список делегатов. Сначала объявляется делегат, а затем переменная события.

public delegate bool NextLineDelegate();

public event NextLineDelegate NextLineEvent; Добавление и исключение методов в списке события осуществляется с помощью операторов += и –=: NextLineEvent += Form1.NextLine; • Наконец, можно сделать вызов через процедурную переменную: NextLineEvent(); • Вызываются все методы, зарегистрированные в списке события

Определяя событие, можно указать методы, которые будут вызываться при добавлении делегата в список и при исключении делегата из списка.



**8)** Мы создаём класс для того, чтобы создавать объекты, так? А классы являются объектами. Метакласс это то, что создаёт эти самые объекты. Они являются классами классов, можно представить это себе следующим образом:

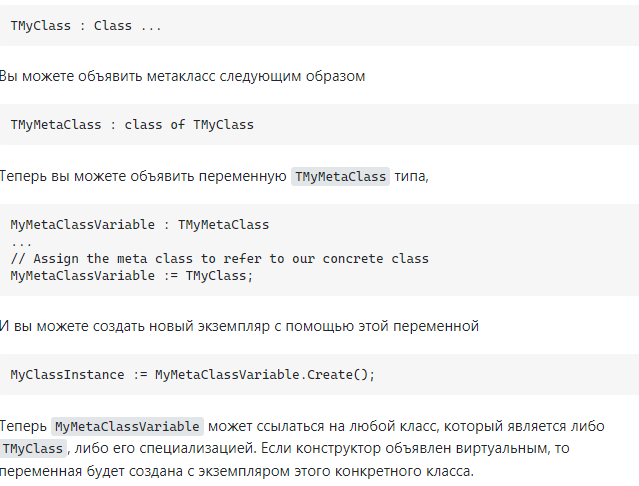
MyClass = MetaClass() - PYTHON

MyObject = MyClass()

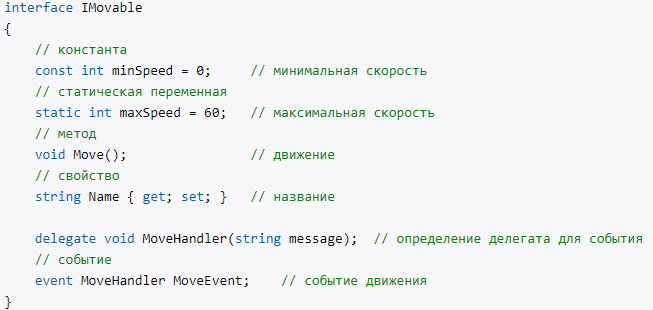
MyClass = type('MyClass', (), {})

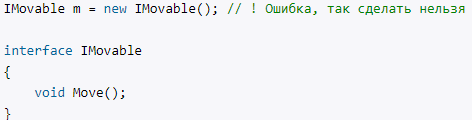
Это потому что функция type на самом деле является метаклассом. type это метакласс, который Питон внутренне использует для создания всех классов.

Единственный язык программирования, с которым я работал и который имеет настоящие виртуальные конструкторы, - это Delphi. В Delphi существует определенный "тип метакласса", то есть конкретная программная конструкция, представляющая метакласс (тогда как в .NET мета-класс, Type class, является просто экземпляром обычного класса). Итак, если у вас есть класс, называемый TMyClass.



**9)** Интерфейс – выделенные в отдельную единицу методы и поля, которые являются общедоступными. При этом методы в интерфейсе по умолчанию неявно являются абстрактными, то есть требуют своей реализации в реализующих их классах. Интерфейс выступает дополнительной точкой доступа к объекту.







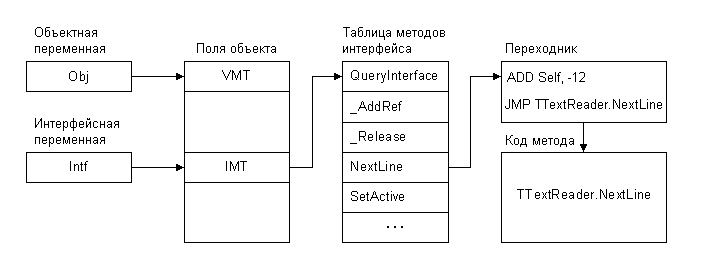
Можно объявить интерфейсную переменную и присвоить ей объект, поддерживающий интерфейс переменной. TableReader reader = new TableReader(...); ITableReader intf = reader; То есть можно одинаково работать с объектами с общим интерфейсом и вызывать методы интерфейса(переопределенные в классе объекта). При этом объект(класс) поддерживает множественное использование интерфейсов, что заменяет множественное наследование. При этом сам интерфейс может быть расширен другим интерфейсом. В этом случае класс, реализующий дочерний интерфейс должен переопределить и методы старшего интерфейса. Интерфейс является особым типом данных: он может быть реализован в одной программе, а использоваться из другой. Для этого нужно обеспечить идентификацию интерфейса при межпрограммном взаимодействии. Понятно, что программный идентификатор интерфейса для этого не подходит — разные программы пишутся разными людьми, а разные люди подчас дают одинаковые имена своим творениям. Поэтому каждому интерфейсу выдается своеобразный «паспорт» — глобально-уникальный идентификатор (Globally Unique Identifier — GUID).

Глобально-уникальный идентификатор — это 16-ти байтовое число, представленное в виде заключенной в фигурные скобки последовательности шестнадцатеричных цифр:

Совместимость интерфейсов подчиняется определенным правилам. Если интерфейс создан расширением уже существующего интерфейса: то интерфейсной переменной базового типа может быть присвоено значение интерфейсной переменной производного типа.

Механизм подсчета ссылок на объект предназначен для автоматического уничтожения неиспользуемых объектов. Неиспользуемым считается объект, на который не ссылается ни одна интерфейсная переменная. Нужно отметить, что объектные переменные не учитываются при подсчете ссылок. Поэтому лучше избегать смешивания интерфейсных и объектных переменных. Если вы планируете использовать объект через интерфейс, то лучше всего результат работы конструктора сразу присвоить интерфейсной переменной

В памяти интерфейс по сути выступает дополнительной таблицей виртуальных методов, ссылка на которую укладывается среди полей объекта. Эта таблица называется *таблицей методов интерфейса*. В ней хранятся указатели на методы класса, реализующие методы интерфейса. Интерфейсная переменная хранит ссылку на скрытое поле объекта, которое содержит указатель на таблицу методов интерфейса. Когда интерфейсной переменной присваивается значение объектой переменной, к адресу объекта добавляется смещение до скрытого поля внутри объекта и этот результат заносится в интерфейсную переменную. Чтобы убедиться в сказанном, посмотрите в отладчике значения Pointer(Obj) и Pointer(Intf) сразу после выполнения оператора Intf := Obj. Эти значения будут разными! Причина в том, что объектная ссылка указывает на начало объекта, а интерфейсная ссылка — на скрытое поле внутри объекта.



Алгоритм вызова метода интерфейса такой же, как алгоритм вызова метода класса. Когда через интерфейсную переменную выполняется вызов метода,

Intf.NextLine;

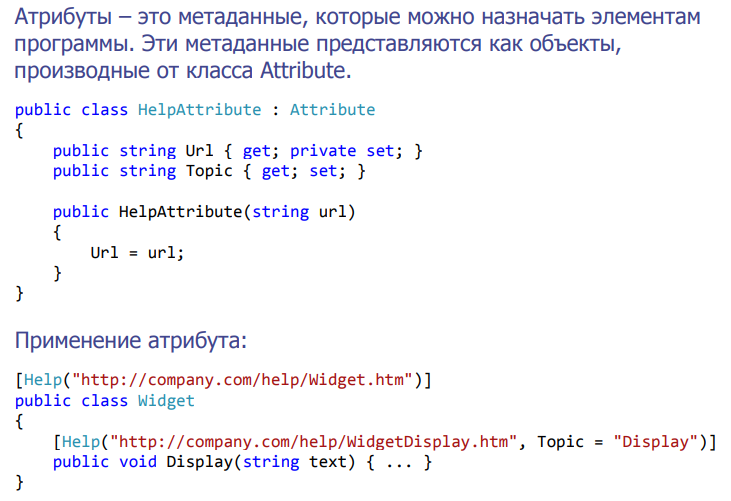
реализуется следующий алгоритм:

1. Из интерфейсной переменной извлекается адрес (по нему хранится адрес таблицы методов интерфейса);
2. По полученному адресу извлекается адрес таблицы методов интерфейса;
3. На основании порядкового номера метода в интерфейсе из таблицы извлекается адрес соответствующей подпрограммы;
4. Вызывается код, находящийся по этому адресу. Этот код является переходником от метода интерфейса к методу объекта. Его задача — восстановить из ссылки на интерфейс значение указателя Self (путем вычитания заранее известного значения) и выполнить прямой переход на код метода класса.

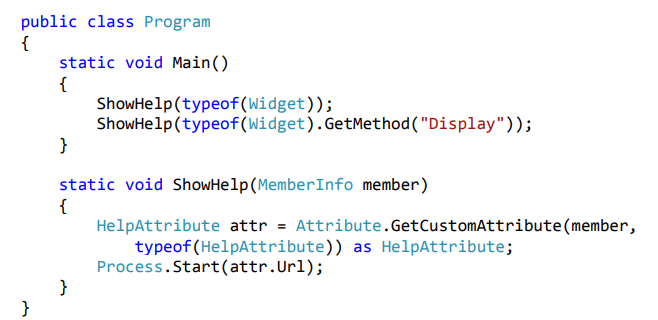
**10)** Можно определить класс, параметризованный типом данных. Такой класс называют шаблоном или обобщенным классом. В java это так и называется: Generics, или обобщения. Обобщенными могут быть как классы, так и методы, в том числе конструктор. Обобщение в java не поддерживает примитивный тип(в отличие от c#). При введении обобщения можно ограничивать диапазон типов. В c# это ключевое слово where:

public class List where T: ListItem { ... public void Add(T item); // Внутри шаблона - List.Add() ... }

В java это extends. Например, в java будет так <T extends Number>. Создание переменных обобщенного типа тоже допускается c#: var stringList = new List(); java: List<String> list = new ArrayList<>(); При этом представление в памяти зависит от типа, который был передан в обобщение.

**11) **

С помощью механизма рефлексии для каждого элемента программы – класса, поля, свойства, метода, параметра – можно получить список атрибутов и воспользоваться данными атрибутов.



Если смотреть на примере языка java, то в java встроено огромное количество аннотаций. Пользовательские аннотации используются крайне редко и считаются плохим тоном:

Предварительно описание аннотации:

@Retention(RetentionPolicy.RUNTIME)

@interface haveArea {

String name(); }

@haveArea(name = "Прямоугольник")

public class Rectangle {

double a;

double b;

public Rectangle(double a, double b) {

this.a = a;

this.b = b;

}

@returnArea

void getArea() {

System.out.println(a \* b);

}

}

Использование:

haveArea annotation = Rectangle.class.getAnnotation(haveArea.class);

System.out.println(annotation.name);

**12)** С делегатами тесно связаны анонимные методы. Анонимные методы используются для создания экземпляров делегатов. Анонимные методы являются усложнением лямбда выражений

Определение анонимных методов начинается с ключевого слова delegate, после которого идет в скобках список параметров и тело метода в фигурных скобках:

delegate(параметры)

{

// инструкции

}

Например:

MessageHandler handler = delegate (string mes)

{

Console.WriteLine(mes);

};

handler("hello world!");

delegate void MessageHandler(string message);

Анонимный метод не может существовать сам по себе, он используется для инициализации экземпляра делегата, как в данном случае переменная handler представляет анонимный метод. И через эту переменную делегата можно вызвать данный анонимный метод.

Существует возможность представить анонимную функцию как данные с помощью шаблона System.Linq.Expressions.Expression:

Func func = (int x) => -x; // код

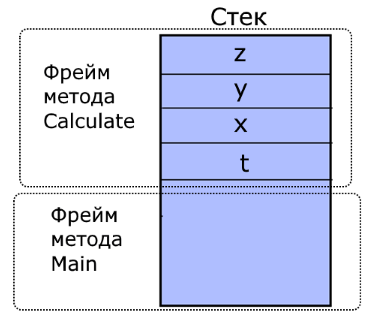
Expression<Func<int, int>> exp = func; // данные

Получение анонимного делегата из сконструированного в переменной типа Expression выражения: Func<int, int> func = exp.Compile(); // теперь код

Вызов: int y = func(10);

Конструируемое выражение не может содержать некоторые элементы языка: блок { } и операторы присваивания.

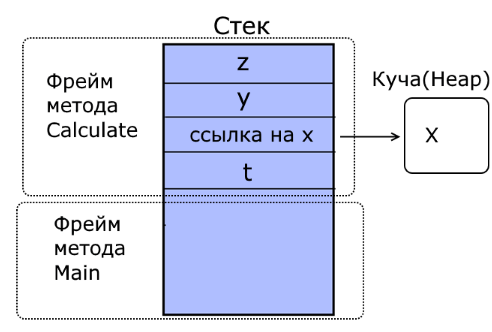
**13)** Чтобы понять разницу между примитивными типами(значения) и ссылочными типами, нужно понимать устройство памяти. Оно одинаково как для c#(.NET), так и для виртуальной машины java. Здесь память делится на два типа: стек и куча (heap). Параметры и переменные метода, которые представляют типы значений, размещают свое значение в стеке. Стек представляет собой структуру данных, которая растет снизу вверх: каждый новый добавляемый элемент помещается поверх предыдущего. Время жизни переменных таких типов ограничено их контекстом. Физически стек - это некоторая область памяти в адресном пространстве. Когда программа только запускается на выполнение, в конце блока памяти, зарезервированного для стека устанавливается указатель стека. При помещении данных в стек указатель переустанавливается таким образом, что снова указывает на новое свободное место. При вызове каждого отдельного метода в стеке будет выделяться область памяти или фрейм стека, где будут храниться значения его параметров и переменных.



При вызове этого метода Calculate в его фрейм в стеке будут помещаться значения t, x, y и z. Они определяются в контексте данного метода. Когда метод отработает, область памяти, которая выделялась под стек, впоследствии может быть использована другими методами. Причем если параметр или переменная метода представляет тип значений, то в стеке будет храниться непосредсвенное значение этого параметра или переменной. Например, в данном случае переменные и параметр метода Calculate представляют значимый тип - тип int, поэтому в стеке будут храниться их числовые значения.

Ссылочные типы хранятся в куче или хипе, которую можно представить как неупорядоченный набор разнородных объектов. Физически это остальная часть памяти, которая доступна процессу.

При создании объекта ссылочного типа в стеке помещается ссылка на адрес в куче (хипе). Когда объект ссылочного типа перестает использоваться, в дело вступает автоматический сборщик мусора: он видит, что на объект в хипе нету больше ссылок, условно удаляет этот объект и очищает память - фактически помечает, что данный сегмент памяти может быть использован для хранения других данных.



Теперь рассмотим ситуацию, когда тип значений и ссылочный тип представляют составные типы - структуру и класс:

State state1 = new State(); // State - структура, ее данные размещены в стеке

Country country1 = new Country();

// Country - класс, в стек помещается ссылка на адрес в хип

// а в хипе располагаются все данные объекта country1

struct State

{

public int x;

public int y;

}

class Country

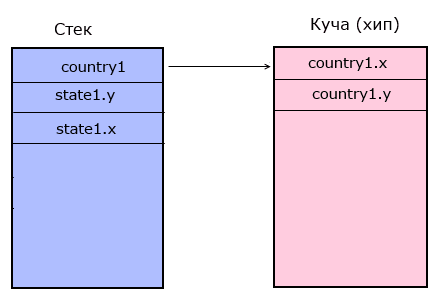
{

public int x;

public int y;

}

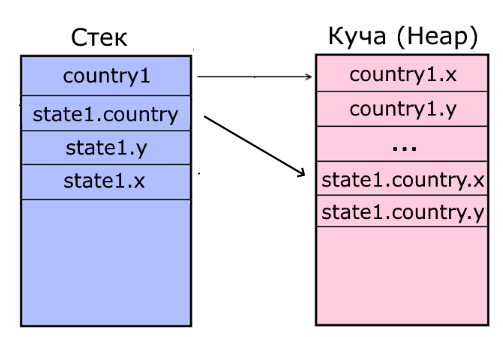
Здесь в методе Main в стеке выделяется память для объекта state1. Далее в стеке создается ссылка для объекта country1 (Country country1), а с помощью вызова конструктора с ключевым словом new выделяется место в хипе (new Country()). Ссылка в стеке для объекта country1 будет представлять адрес на место в хипе, по которому размещен данный объект..



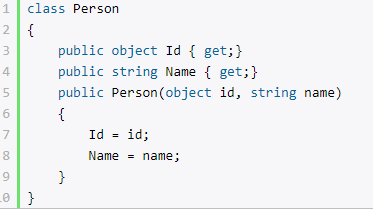
Таким образом, в стеке окажутся все поля структуры state1 и ссылка на объект country1 в хипе.

Но, допустим, в структуре State также определена переменная ссылочного типа Country.

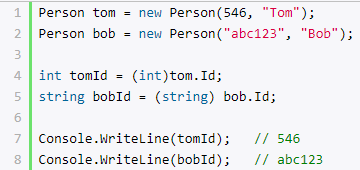
Значение переменной state1.country также будет храниться в куче, так как эта переменная представляет ссылочный тип:



Упаковка и распаковка:



Затем этот класс можно было использовать для создания пользователей в программе:



Все вроде замечательно работает, но такое решение является не очень оптимальным. Дело в том, что в данном случае мы сталкиваемся с такими явлениями как упаковка (boxing) и распаковка (unboxing).

Так, при передаче в конструктор значения типа int, происходит упаковка этого значения в тип Object: Так как тип object является универсальным типом, от которого наследуется все типы, соответственно в свойствах подобного типа мы можем сохранить и строки, и числа.

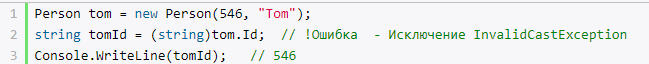
Person tom = new Person(546, "Tom"); // упаковка в значения int в тип Object

Чтобы обратно получить данные в переменную типов int, необходимо выполнить распаковку:

int tomId = (int)tom.Id; // Распаковка в тип int

Упаковка (boxing) предполагает преобразование объекта значимого типа (например, типа int) к типу object. При упаковке общеязыковая среда CLR обертывает значение в объект типа System.Object и сохраняет его в управляемой куче (хипе). Распаковка (unboxing), наоборот, предполагает преобразование объекта типа object к значимому типу. Упаковка и распаковка ведут к снижению производительности, так как системе надо осуществить необходимые преобразования.

Кроме того, существует другая проблема - проблема безопасности типов. Так, мы получим ошибку во время выполнения программы, если напишем следующим образом:



Эту проблему решают обобщения.

Кроме стандартных значений типа чисел, строк, язык C# имеет специальное значение - null, которое фактически указывает на отсутствие значения как такового, отсутствие данных. До сих пор значение null выступает как значение по умолчанию для ссылочных типов. Но начиная с версии C# 8.0 в язык была введена концепция ссылочных nullable-типов (nullable reference types) и nullable aware context - nullable-контекст, в котором можно использовать ссылочные nullable-типы. Чтобы определить переменную/параметр ссылочного типа, как переменную/параметр, которым можно присваивать значение null, после названия типа указывается знак вопроса ?



Зачем нужно это значение null? В различных ситуациях бывает удобно, чтобы объекты могли принимать значение null, то есть были бы не определены. Стандартный пример - работа с базой данных, которая может содержать значения null. И мы можем заранее не знать, что мы получим из базы данных - какое-то определенное значение или же null.

При этом подобные ссылочные типы, которые допускают присвоение значения null, доступно только в nullable-контексте. Для nullable-контекста характерны следующие особенности:

Переменную ссылочного типа следует инициализировать конкретным значением, ей не следует присваивать значение null

Переменной ссылочного nullable-типа можно присвоить значение null, но перед использованием необходимо проверять ее на значение null.

**14)** см, предыдущие вопросы, если нужно.

**15)** **Визуальное программирование** — способ создания программы для ЭВМ путём манипулирования графическими объектами вместо написания её текста. Рассматривая системы визуального программирования, легко увидеть, что все они базируются на объектно-ориентированном программировании и являются его логическим продолжением.

Компонента представляет собой разновидность объекта, который можно перенести (агрегировать) в приложение. Компоненты бывают визуальными и невизуальными. Первые предназначены для организации интерфейса с пользователем. Это различные кнопки, списки, статический и редактируемый текст, изображения и многое другое. Эти компоненты отображаются при выполнении разрабатываемого приложения. Невизуальные компоненты отвечают за доступ к системным ресурсам: драйверам баз данных, таймерам и т. д. Во время разработки они отображаются своей пиктограммой, но при выполнении приложения, как правило, невидимы. Компонента может принадлежать либо другой компоненте, либо форме. Формой называется визуальная компонента, обладающая свойством окна. На сегодняшний день существует множество библиотек компонентов, например, для React: Ant Design, Mantine, Material UI.

Расширяемая программа – программа, которую можно развивать лишь дополняя ее новыми модулями, не изменяя старых.

Расширяемое программирование – возможность конструирования таких иерархий модулей, когда каждый модуль добавляет новую функциональность в систему. Расширяемое программирование подразумевает, что добавление модуля возможно без необходимости вносить какие-либо изменения в существующие модули — не должно быть необходимости даже их перекомпилировать. Новые модули не только добавляют новые процедуры, но, что более важно, добавляют также новые (расширенные) типы данных.

Использование объектно-ориентированного программирования позволяет писать более структурированный и читабельный код, что способствует построению расширяемых программ.

**16)** **SOLID** - это принципы разработки программного обеспечения, следуя которым Вы получите хороший код, который в дальнейшем будет хорошо масштабироваться и поддерживаться в рабочем состоянии.

**S - Single Responsibility Principle - принцип единственной ответственности**. Каждый класс должен иметь только одну зону ответственности.

**O - Open closed Principle - принцип открытости-закрытости.** Классы должны быть открыты для расширения, но закрыты для изменения.

**L - Liskov substitution Principle - принцип подстановки Барбары Лисков.** Должна быть возможность вместо базового (родительского) типа (класса) подставить любой его подтип (класс-наследник), при этом работа программы не должна измениться.

**I -  Interface Segregation Principle - принцип разделения интерфейсов.** Данный принцип обозначает, что не нужно заставлять клиента (класс) реализовывать интерфейс, который не имеет к нему отношения.

**D - Dependency Inversion Principle - принцип инверсии зависимостей.** Модули верхнего уровня не должны зависеть от модулей нижнего уровня. И те, и другие должны зависеть от абстракции. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.\

Про SOLID - https://habr.com/ru/articles/688530/

Про SOA - <https://habr.com/ru/companies/vk/articles/342526/>

**17)** **Итератор** — это поведенческий паттерн проектирования, который даёт возможность последовательно обходить элементы составных объектов, не раскрывая их внутреннего представления.

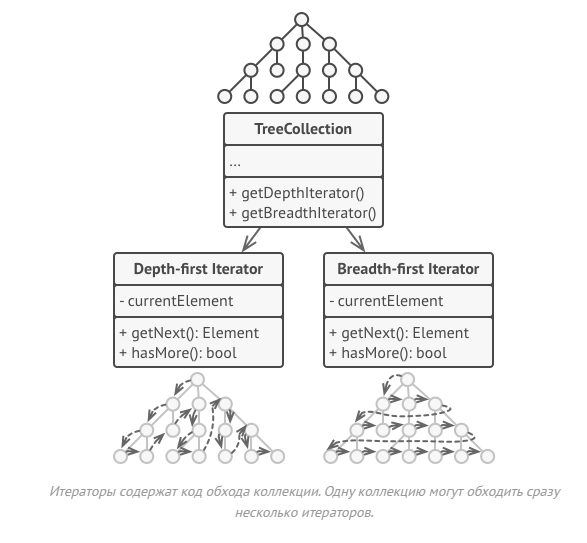
**Проблема:** Как бы ни была структурирована коллекция, пользователь должен иметь возможность последовательно обходить её элементы, чтобы проделывать с ними какие-то действия.

Но каким способом следует перемещаться по сложной структуре данных? Например, сегодня может быть достаточным обход дерева в глубину, но завтра потребуется возможность перемещаться по дереву в ширину. А на следующей неделе и того хуже — понадобится обход коллекции в случайном порядке.

**Решение:** Идея паттерна Итератор состоит в том, чтобы вынести поведение обхода коллекции из самой коллекции в отдельный класс.

Объект-итератор будет отслеживать состояние обхода, текущую позицию в коллекции и сколько элементов ещё осталось обойти. Одну и ту же коллекцию смогут одновременно обходить различные итераторы, а сама коллекция не будет даже знать об этом.

К тому же, если вам понадобится добавить новый способ обхода, вы сможете создать отдельный класс итератора, не изменяя существующий код коллекции.





**18)** **Одиночка** — это порождающий паттерн проектирования, который гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

**Проблема:** Одиночка решает сразу две проблемы, нарушая принцип единственной ответственности класса.

1. **Гарантирует наличие единственного экземпляра класса**. Чаще всего это полезно для доступа к какому-то общему ресурсу, например, базе данных.

Представьте, что вы создали объект, а через некоторое время пробуете создать ещё один. В этом случае хотелось бы получить старый объект, вместо создания нового.

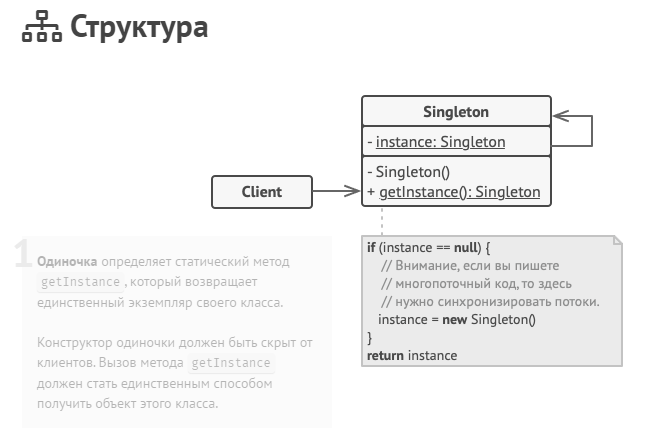
Такое поведение невозможно реализовать с помощью обычного конструктора, так как конструктор класса **всегда** возвращает новый объект.

1. **Предоставляет глобальную точку доступа**. Это не просто глобальная переменная, через которую можно достучаться к определённому объекту. Глобальные переменные не защищены от записи, поэтому любой код может подменять их значения без вашего ведома.

Но есть и другой нюанс. Неплохо бы хранить в одном месте и код, который решает проблему №1, а также иметь к нему простой и доступный интерфейс.

**Решение:** Все реализации одиночки сводятся к тому, чтобы скрыть конструктор по умолчанию и создать публичный статический метод, который и будет контролировать жизненный цикл объекта-одиночки.

Если у вас есть доступ к классу одиночки, значит, будет доступ и к этому статическому методу. Из какой точки кода вы бы его ни вызвали, он всегда будет отдавать один и тот же объект.



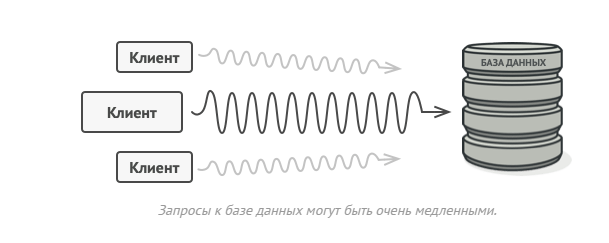
**19) Заместитель** — это структурный паттерн проектирования, который позволяет подставлять вместо реальных объектов специальные объекты-заменители. Эти объекты перехватывают вызовы к оригинальному объекту, позволяя сделать что-то до или после передачи вызова оригиналу.

**Проблема:** Для чего вообще контролировать доступ к объектам? Рассмотрим такой пример: у вас есть внешний ресурсоёмкий объект, который нужен не все время, а изредка. Мы могли бы создавать этот объект не в самом начале программы, а только тогда, когда он кому-то реально понадобится. Каждый клиент объекта получил бы некий код отложенной инициализации. Но, вероятно, это привело бы к множественному дублированию кода.

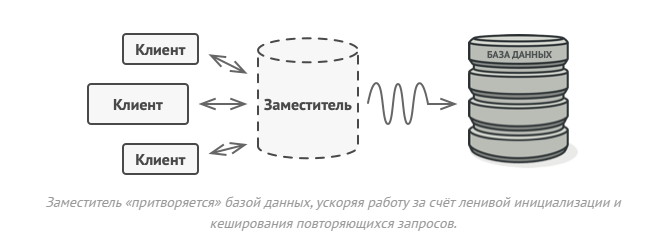
В идеале, этот код хотелось бы поместить прямо в служебный класс, но это не всегда возможно. Например, код класса может находиться в закрытой сторонней библиотеке.

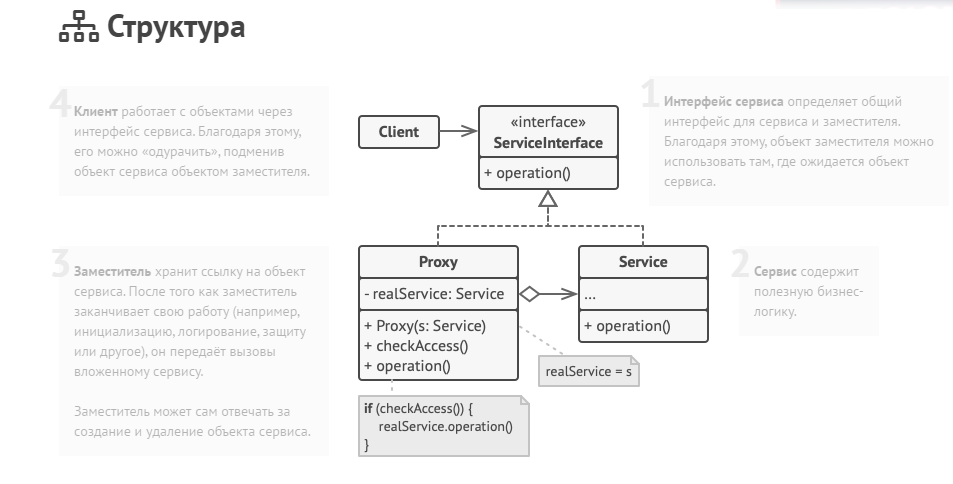
**Решение:** Паттерн Заместитель предлагает создать новый класс-дублёр, имеющий тот же интерфейс, что и оригинальный служебный объект. При получении запроса от клиента объект-заместитель сам бы создавал экземпляр служебного объекта и переадресовывал бы ему всю реальную работу. Но в чём же здесь польза? Вы могли бы поместить в класс заместителя какую-то промежуточную логику, которая выполнялась бы до (или после) вызовов этих же методов в настоящем объекте. А благодаря одинаковому интерфейсу, объект-заместитель можно передать в любой код, ожидающий сервисный объект.

Было:



Стало:





**20) Компоновщик** — это структурный паттерн проектирования, который позволяет сгруппировать множество объектов в древовидную структуру, а затем работать с ней так, как будто это единичный объект.

**Проблема:** Паттерн Компоновщик имеет смысл только тогда, когда основная модель вашей программы может быть структурирована в виде дерева.

Например, есть два объекта: Продукт и Коробка. Коробка может содержать несколько Продуктов и других Коробок поменьше. Те, в свою очередь, тоже содержат либо Продукты, либо Коробки и так далее.

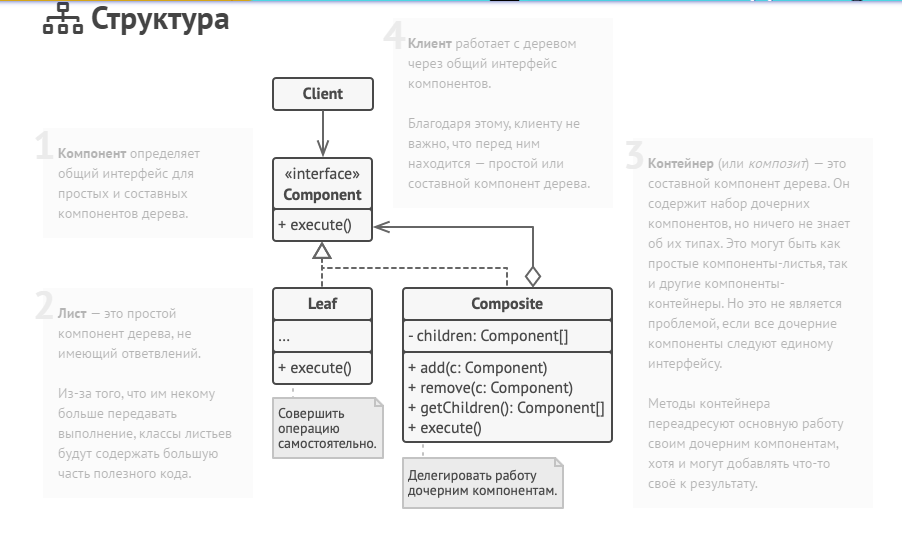
Теперь предположим, ваши Продукты и Коробки могут быть частью заказов. Каждый заказ может содержать как простые Продукты без упаковки, так и составные Коробки. Ваша задача состоит в том, чтобы узнать цену всего заказа.

Если решать задачу в лоб, то вам потребуется открыть все коробки заказа, перебрать все продукты и посчитать их суммарную стоимость. Но это слишком хлопотно, так как типы коробок и их содержимое могут быть вам неизвестны. Кроме того, наперёд неизвестно и количество уровней вложенности коробок, поэтому перебрать коробки простым циклом не выйдет.

**Решение:** Компоновщик предлагает рассматривать Продукт и Коробку через единый интерфейс с общим методом получения стоимости.

Продукт просто вернёт свою цену. Коробка спросит цену каждого предмета внутри себя и вернёт сумму результатов. Если одним из внутренних предметов окажется коробка поменьше, она тоже будет перебирать своё содержимое, и так далее, пока не будут посчитаны все составные части.

Для вас, клиента, главное, что теперь не нужно ничего знать о структуре заказов. Вы вызываете метод получения цены, он возвращает цифру, а вы не тонете в горах картона и скотча.



**21) Мост** — это структурный паттерн проектирования, который разделяет один или несколько классов на две отдельные иерархии — абстракцию и реализацию, позволяя изменять их независимо друг от друга.

**Проблема:** Абстракция? Реализация?! Звучит пугающе! Чтобы понять, о чём идёт речь, давайте разберём очень простой пример.

У вас есть класс геометрических Фигур, который имеет подклассы Круг и Квадрат. Вы хотите расширить иерархию фигур по цвету, то есть иметь Красные и Синие фигуры. Но чтобы всё это объединить, вам придётся создать 4 комбинации подклассов, вроде СиниеКруги и КрасныеКвадраты.

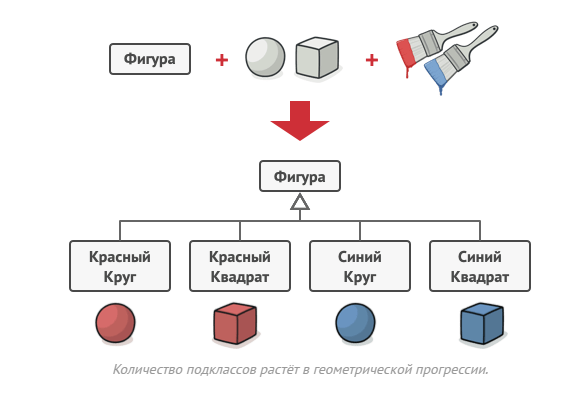
При добавлении новых видов фигур и цветов количество комбинаций будет расти в геометрической прогрессии. Например, чтобы ввести в программу фигуры треугольников, придётся создать сразу два новых подкласса треугольников под каждый цвет. После этого новый цвет потребует создания уже трёх классов для всех видов фигур. Чем дальше, тем хуже.

**Решение:** Корень проблемы заключается в том, что мы пытаемся расширить классы фигур сразу в двух независимых плоскостях — по виду и по цвету. Именно это приводит к разрастанию дерева классов.

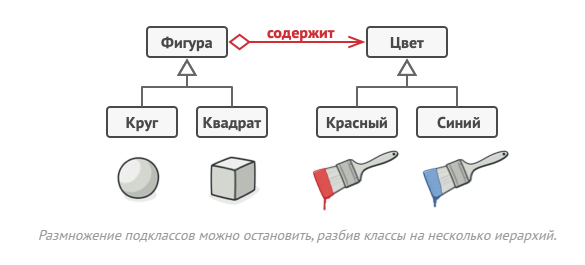
Паттерн Мост предлагает заменить наследование агрегацией или композицией. Для этого нужно выделить одну из таких «плоскостей» в отдельную иерархию и ссылаться на объект этой иерархии, вместо хранения его состояния и поведения внутри одного класса.

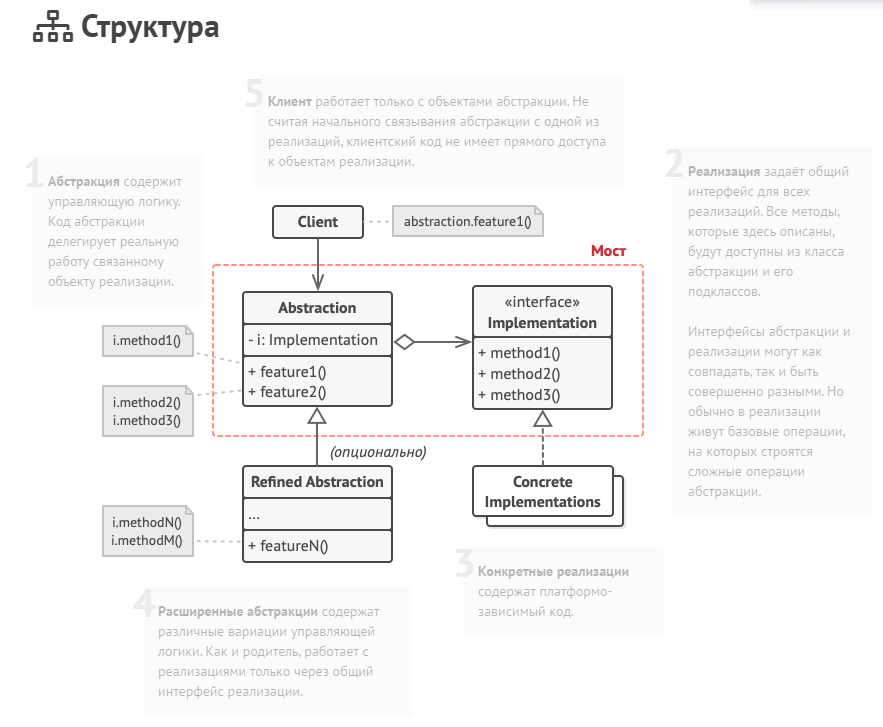
Таким образом, мы можем сделать Цвет отдельным классом с подклассами Красный и Синий. Класс Фигур получит ссылку на объект Цвета и сможет делегировать ему работу, если потребуется. Такая связь и станет мостом между Фигурами и Цветом. При добавлении новых классов цветов не потребуется трогать классы фигур и наоборот.

Было:



Стало:





**22)** **Наблюдатель** — это поведенческий паттерн проектирования, который создаёт механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах.

**Проблема:**

Представьте, что вы имеете два объекта: Покупатель и Магазин. В магазин вот-вот должны завезти новый товар, который интересен покупателю.

Покупатель может каждый день ходить в магазин, чтобы проверить наличие товара. Но при этом он будет злиться, без толку тратя своё драгоценное время.

С другой стороны, магазин может разослать спам каждому своему покупателю. Многих это расстроит, так как товар специфический, и не всем он нужен.

Получается конфликт: либо покупатель тратит время на периодические проверки, либо магазин тратит ресурсы на бесполезные оповещения.

**Решение:**

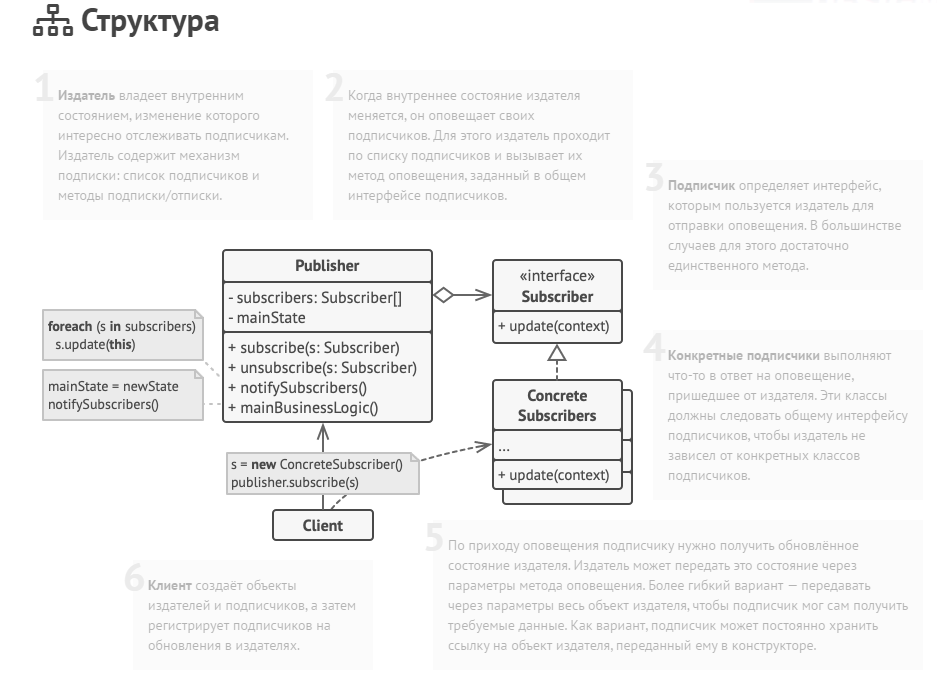
Давайте называть Издателями те объекты, которые содержат важное или интересное для других состояние. Остальные объекты, которые хотят отслеживать изменения этого состояния, назовём Подписчиками.

Паттерн Наблюдатель предлагает хранить внутри объекта издателя список ссылок на объекты подписчиков, причём издатель не должен вести список подписки самостоятельно. Он предоставит методы, с помощью которых подписчики могли бы добавлять или убирать себя из списка.

Теперь самое интересное. Когда в издателе будет происходить важное событие, он будет проходиться по списку подписчиков и оповещать их об этом, вызывая определённый метод объектов-подписчиков.

Издателю безразлично, какой класс будет иметь тот или иной подписчик, так как все они должны следовать общему интерфейсу и иметь единый метод оповещения.

Увидев, как складно всё работает, вы можете выделить общий интерфейс, описывающий методы подписки и отписки, и для всех издателей. После этого подписчики смогут работать с разными типами издателей, а также получать оповещения от них через один и тот же метод.



**23) Посетитель** — это поведенческий паттерн проектирования, который позволяет добавлять в программу новые операции, не изменяя классы объектов, над которыми эти операции могут выполняться.

**Проблема:**

Ваша команда разрабатывает приложение, работающее с геоданными в виде графа. Узлами графа являются городские локации: памятники, театры, рестораны, важные предприятия и прочее. Каждый узел имеет ссылки на другие, ближайшие к нему узлы. Каждому типу узлов соответствует свой класс, а каждый узел представлен отдельным объектом.

Ваша задача — сделать экспорт этого графа в XML. Дело было бы плёвым, если бы вы могли редактировать классы узлов. Достаточно было бы добавить метод экспорта в каждый тип узла, а затем, перебирая узлы графа, вызывать этот метод для каждого узла. Благодаря полиморфизму, решение получилось бы изящным, так как вам не пришлось бы привязываться к конкретным классам узлов.

Но, к сожалению, классы узлов вам изменить не удалось. Системный архитектор сослался на то, что код классов узлов сейчас очень стабилен, и от него многое зависит, поэтому он не хочет рисковать и позволять кому-либо его трогать.

К тому же он сомневался в том, что экспорт в XML вообще уместен в рамках этих классов. Их основная задача была связана с геоданными, а экспорт выглядит в рамках этих классов чужеродно.

Была и ещё одна причина запрета. Если на следующей неделе вам бы понадобился экспорт в какой-то другой формат данных, то эти классы снова пришлось бы менять.

**Решение:**

Паттерн Посетитель предлагает разместить новое поведение в отдельном классе, вместо того чтобы множить его сразу в нескольких классах. Объекты, с которыми должно было быть связано поведение, не будут выполнять его самостоятельно. Вместо этого вы будете передавать эти объекты в методы посетителя.

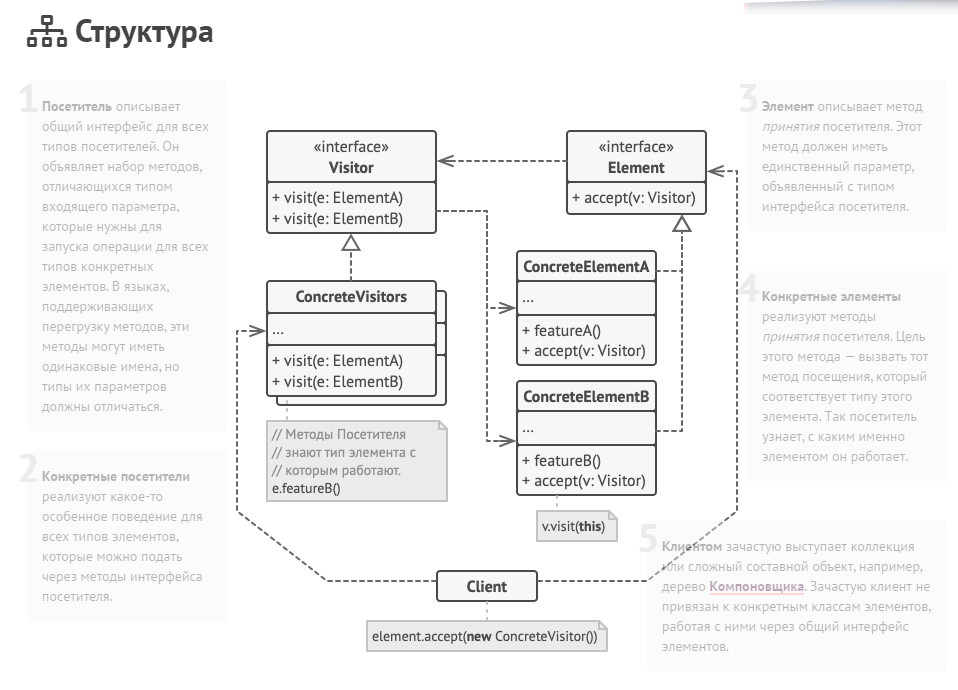
Код поведения, скорее всего, должен отличаться для объектов разных классов, поэтому и методов у посетителя должно быть несколько. Названия и принцип действия этих методов будет схож, но основное отличие будет в типе принимаемого в параметрах объекта.

Здесь возникает вопрос: как подавать узлы в объект-посетитель? Так как все методы имеют отличающуюся сигнатуру, использовать полиморфизм при переборе узлов не получится. Придётся проверять тип узлов для того, чтобы выбрать соответствующий метод посетителя.

Тут не поможет даже механизм перегрузки методов (доступный в Java и C#). Если назвать все методы одинаково, то неопределённость реального типа узла всё равно не даст вызвать правильный метод. Механизм перегрузки всё время будет вызывать метод посетителя, соответствующий типу Node, а не реального класса поданного узла.

Но паттерн Посетитель решает и эту проблему, используя механизм [**двойной диспетчеризации**](https://refactoring.guru/ru/design-patterns/visitor-double-dispatch). Вместо того, чтобы самим искать нужный метод, мы можем поручить это объектам, которые передаём в параметрах посетителю. А они уже вызовут правильный метод посетителя.

Как видите, изменить классы узлов всё-таки придётся. Но это простое изменение позволит применять к объектам узлов и другие поведения, ведь классы узлов будут привязаны не к конкретному классу посетителей, а к их общему интерфейсу. Поэтому если придётся добавить в программу новое поведение, вы создадите новый класс посетителей и будете передавать его в методы узлов.



**24)** **Фабричный метод** — это порождающий паттерн проектирования, который определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов.

**Проблема:**

Представьте, что вы создаёте программу управления грузовыми перевозками. Сперва вы рассчитываете перевозить товары только на автомобилях. Поэтому весь ваш код работает с объектами класса Грузовик.

В какой-то момент ваша программа становится настолько известной, что морские перевозчики выстраиваются в очередь и просят добавить поддержку морской логистики в программу.

Отличные новости, правда?! Но как насчёт кода? Большая часть существующего кода жёстко привязана к классам Грузовиков. Чтобы добавить в программу классы морских Судов, понадобится перелопатить всю программу. Более того, если вы потом решите добавить в программу ещё один вид транспорта, то всю эту работу придётся повторить.

В итоге вы получите ужасающий код, наполненный условными операторами, которые выполняют то или иное действие, в зависимости от класса транспорта.

**Решение:**

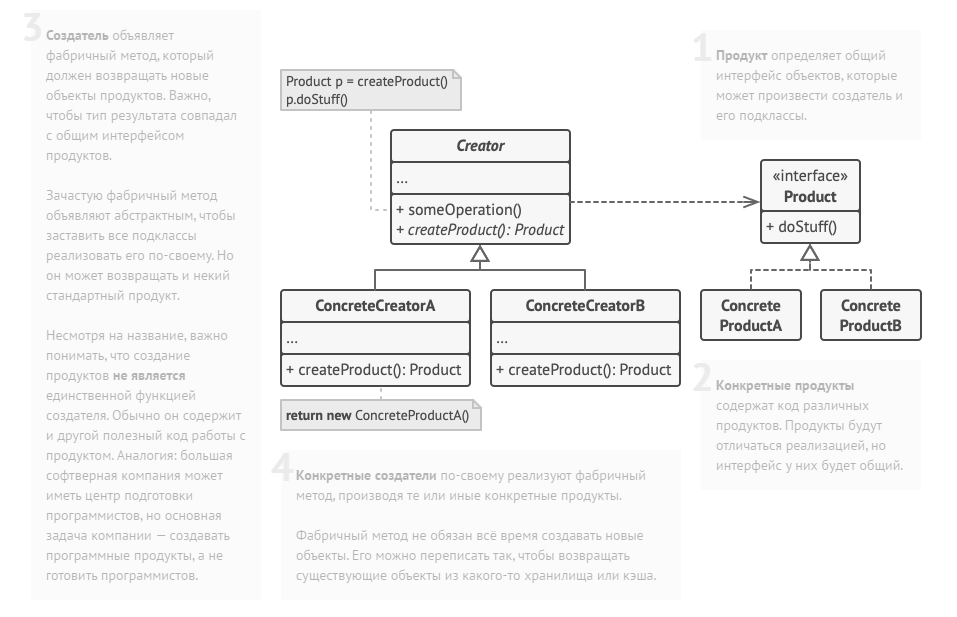
Паттерн Фабричный метод предлагает создавать объекты не напрямую, используя оператор new, а через вызов особого фабричного метода. Не пугайтесь, объекты всё равно будут создаваться при помощи new, но делать это будет фабричный метод.

На первый взгляд, это может показаться бессмысленным: мы просто переместили вызов конструктора из одного конца программы в другой. Но теперь вы сможете переопределить фабричный метод в подклассе, чтобы изменить тип создаваемого продукта.

Чтобы эта система заработала, все возвращаемые объекты должны иметь общий интерфейс. Подклассы смогут производить объекты различных классов, следующих одному и тому же интерфейсу.

Например, классы Грузовик и Судно реализуют интерфейс Транспорт с методом доставить. Каждый из этих классов реализует метод по-своему: грузовики везут грузы по земле, а суда — по морю. Фабричный метод в классе ДорожнойЛогистики вернёт объект-грузовик, а класс МорскойЛогистики — объект-судно.

Для клиента фабричного метода нет разницы между этими объектами, так как он будет трактовать их как некий абстрактный Транспорт. Для него будет важно, чтобы объект имел метод доставить, а как конкретно он работает — не важно.



**Фабрика классов(абстрактная фабрика) -** — это порождающий паттерн проектирования, который позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов.

**Проблема:**

Представьте, что вы пишете симулятор мебельного магазина. Ваш код содержит:

1. Семейство зависимых продуктов. Скажем, Кресло + Диван + Столик.
2. Несколько вариаций этого семейства. Например, продукты Кресло, Диван и Столик представлены в трёх разных стилях: Ар-деко, Викторианском и Модерне.

Вам нужен такой способ создавать объекты продуктов, чтобы они сочетались с другими продуктами того же семейства. Это важно, так как клиенты расстраиваются, если получают несочетающуюся мебель.

Кроме того, вы не хотите вносить изменения в существующий код при добавлении новых продуктов или семейcтв в программу. Поставщики часто обновляют свои каталоги, и вы бы не хотели менять уже написанный код каждый раз при получении новых моделей мебели.

**Решение:**

Для начала паттерн Абстрактная фабрика предлагает выделить общие интерфейсы для отдельных продуктов, составляющих семейства. Так, все вариации кресел получат общий интерфейс Кресло, все диваны реализуют интерфейс Диван и так далее.

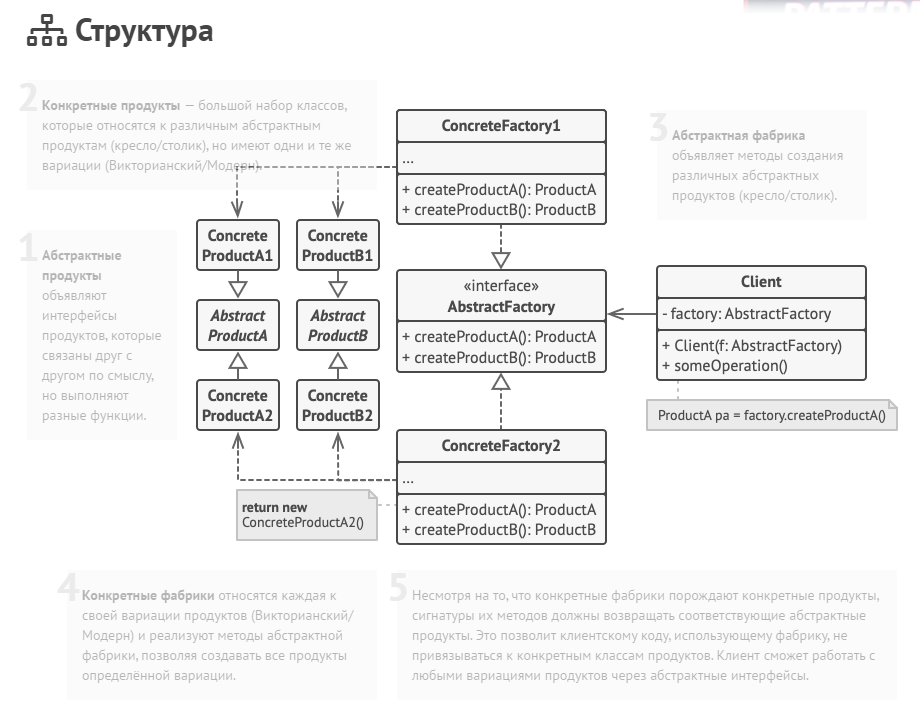
Далее вы создаёте абстрактную фабрику — общий интерфейс, который содержит методы создания всех продуктов семейства (например, создатьКресло, создатьДиван и создатьСтолик). Эти операции должны возвращать **абстрактные** типы продуктов, представленные интерфейсами, которые мы выделили ранее — Кресла, Диваны и Столики.

Как насчёт вариаций продуктов? Для каждой вариации семейства продуктов мы должны создать свою собственную фабрику, реализовав абстрактный интерфейс. Фабрики создают продукты одной вариации. Например, ФабрикаМодерн будет возвращать только КреслаМодерн,ДиваныМодерн и СтоликиМодерн.

Клиентский код должен работать как с фабриками, так и с продуктами только через их общие интерфейсы. Это позволит подавать в ваши классы любой тип фабрики и производить любые продукты, ничего не ломая.

Например, клиентский код просит фабрику сделать стул. Он не знает, какого типа была эта фабрика. Он не знает, получит викторианский или модерновый стул. Для него важно, чтобы на стуле можно было сидеть и чтобы этот стул отлично смотрелся с диваном той же фабрики.

Осталось прояснить последний момент: кто создаёт объекты конкретных фабрик, если клиентский код работает только с интерфейсами фабрик? Обычно программа создаёт конкретный объект фабрики при запуске, причём тип фабрики выбирается, исходя из параметров окружения или конфигурации.



**26)** В языке С++ очень бедные средства модульного программирования, поэтому для достижения модульности программ, следует придерживаться определенных принципов.

Роль программного интерфейса модуля играет h-файл, а cpp-файл — роль реализации этогомодуля. Внутрь h-файла включаются h-файлы других модулей, необходимые для компиляции интерфейсной части. Внутрь cpp-файла включаются h-файлы других модулей,необходимые для компиляции cpp- и h-файлов интерфейсной части модуля.

В больших проектах наблюдается серьезная проблема — конфликт идентификаторов. Она решается с помощью пространства имен.

namespaceSys

{

int var;

void Proc();

}

Внутри пространства имен обращение к определенным внутри переменным и

подпрограммам можно осуществлять, используя неполную форму записи:

var = 10;

Proc();

за пределами – надо использовать полную форму записи:

Sys::var = 10;

Sys::Proc();

Для того чтобы избежать возможного конфликта идентификаторов, все определения внутримодуля следует помещать в пространство имен. Следует давать небольшой буквенный идентификатор, который будет соответствовать префиксу файла**.**

**27)** Классы в С++ определяются с помощью одного из ключевых слов: class или struct.

В С++ доступны атрибуты доступа в классах:

-public

-protected

-private

Наследование класса выполняется следующим образом:

class TDelimitedReader: public TTextReader

{

...

};

При наследовании указываются атрибуты доступа к элементам базового класса (public, protected, private). Для того чтобы понять смысл атрибута доступа к базовому классу, базовый класс следует рассматривать, как поле производного класса.

Конструктор создает объект и инициализирует память для него (деструктор — наоборот).

classTTextReader

{

public:

TTextReader();

~TTextReader();

};

В С++, если планируется создавать объекты в динамической памяти (по ссылке),

деструктор необходимо делать виртуальным:

virtual ~TTextReader();

Создание объектов:

-по значению (на стеке):

TtextReader reader{params};

-по ссылке (в динамической памяти):

TTextReader\* reader = new TtextReader{params}; //оператор new служит для размещения объекта в динамической памяти

Таким способом объект создается по ссылке по указанному адресу памяти.

Разрушение объектов:

- если объект создан по значению (на стеке), его разрушение выполняется

автоматически при выходе переменной за область видимости

{

TtextReader reader;

...

} // здесь происходит разрушение объекта reader при автоматическом вызоведеструктора

-если объект создан в динамической памяти (по ссылке), он должен быть уничтожен с помощью оператора delete:

delete reader;

При этом сначала происходит вызов деструктора, а затем — освобождение динамической памяти.

Если программист не определяет в классе конструкторы, то компилятор создает автоматически два конструктора:

-конструктор без параметров

-конструктор копирования

Пример:

classTTextReader

{

public:

TTextReader(); // конструктор без параметров

TTextReader(const TTextReader& reader); // конструктор копирования

}

Внимание! Если программист определил хотя бы один конструктор в класс — компилятор несоздаст никаких стандартных конструкторов.Конструктор без параметров создается для того, чтобы можно было написать:

TTextReader R{};

Конструктор копирования нужен для следующей записи:

TTextReader R1 = R2; // означаетTTextReaderR1(R2);

Конструктор копирования вызывается в том случае, когда создаваемый по значению объектсоздается путем копирования другого уже существующего объекта.Следует отметить, что запись:

TTextReader R1 = R2;

и два оператора:

TTextReader R1;

R1 = R2;

имеют схожий синтаксис с вызовом конструктора копирования, но разную семантику: в

первом случае объект создается конструктором копирования, во втором — конструктором без параметров, а затем с помощью оператора ‘=’ выполняется присваивание одного объектадругому (данный вариант требует перегрузки оператора ‘=’ для класса TTextReader). Работа стандартного конструктора копирования, создаваемого компилятором, заключается втом, чтобы выполнить полное копирование памяти с помощью функции memcpy.

В конструкторе производного класса конструктор базового класса вызывается автоматически до выполнения первого оператора в теле конструктора.В деструкторе производного класса деструктор базового класса вызывается автоматическипосле последнего оператора в теле деструктора.Если базовый класс содержит конструктор с параметрами или несколько конструкторов, товозникает неопределенность в том, какой конструктор базового класса будет вызван, тогда конструктор родительского класса вызывается так:

Child(params): Parent(params)  
{

}.

Если поле объявлено с ключевым словом static, то это — обычная глобальная переменная,для которой имя класса используется как пространство имен.

Если метод объявляется с этим словом, то это — обычная глобальная функция, которая является другом класса. Такая функция не имеет псевдо-параметра this.

**28)** В С++ отсутствует аналог блока try…finally…end.

На платформе Windows благодаря структурной обработке ОС существуют следующий блок:

\_\_try

{

...

}

\_\_finally

{

...

}

Но следует отметить, что для переносимых программ он не подходит. Обработка исключений в c++ имеет следующий вид:

**try**

**{**

**...**

**}**

**catch(std::ios\_base::failure)**

**{**

**...**

**}**

**catch(std::exception)**

**{**

**...**

**}**

**catch(...)**

**{**

**...**

**}**

Распознавание исключительных ситуаций происходит последовательно блоками catch, поэтому их последовательность должна быть от частного к общему. Последний блок catch в примере выше ловит любую исключительную ситуацию. Создание исключительных ситуаций выполняется с помощью оператора throw (аналог raise в Delphi):

throw std::exception("Ошибка");

Внутри блока catch оператор throw возобновляет исключительную ситуацию, как и raise в Delphi. При создании исключительной ситуации при помощи оператора throw объект, описывающий исключительную ситуацию, может быть создан в динамической памяти:

throw new std::exception("Ошибка");

Если применяется такой способ создания исключительной ситуации, ее уничтожение должно происходить следующим образом:

try

{

...

throw new std::exception("Ошибка");

}

catch(std::exception \*e)

{

delete e;

}

catch(...)

{

...

}

Если же записать так:

try

{

...

throw new std::exception("Ошибка");

}

catch(...)

{

...

}

то возникнет утечка ресурсов из-за того, что объект std::exception, созданный в

динамической памяти, не будет освобожден. В С++ отсутствует общий базовый класс для исключительных ситуаций, поэтому на верхнем уровне работы программы нужно отлавливать все возможные базовые классы исключительных ситуаций. Это является препятствием на пути построения расширяемых систем.

Поскольку в С+ отсутствует блок try…finally, его приходится эмулировать.

Object \*p = new Object();

try

{

...

}

catch(...)

{

delete p;

throw;

}

delete p;

Данный код эквивалентен следующему:

Object \*p = new Object();

\_\_try

{

...

}

\_\_finally

{

delete p;

}

за исключением того, что второй пример не является переносимым. Согласно стандарту С++ в деструкторах и операторах delete не должно быть исключительных ситуаций, если же исключительная ситуация произошла, то поведение программы не определено. Если исключительная ситуация происходит в конструкторе объекта, объект считается не созданным и деструктор для этого объекта не вызывается, но память, выделенная для

объекта, освобождается. Если внутри объекта агрегированы другие объекты, то вызываются деструкторы лишь для тех объектов, которые были полностью созданы к моменту возникновения исключительной ситуации. Если объект создается в динамической памяти и освобождается в той же самой процедуре, то для защиты от утечки ресурсов можно применять оболочечные объекты — wrapper (содержит указатель на динамический объект, который уничтожается в деструкторе оболочечного объекта). Оболочечный элемент создается на стеке, поэтому его деструктор вызывается автоматически, гарантируя тем самым уничтожение агрегированного динамического объекта. Такие оболочечные объекты в библиотеках программирования называются AutoPtr, SafePtr и т.д.

**29)** В С++ поддерживается множественное наследование. В этом случае конструктор базовых классов вызывается автоматически в порядке их упоминания в описании класса. Деструктор же базовых классов вызывается строго в обратном порядке.Каждый конструктор перед началом своей работы инициализирует указатель vtable (в Delphi он называется VMT). Конструктор базового класса тоже инициализирует этот указатель. В результате этого объект как бы "рождается", сначала становясь экземпляром базового класса,а затем производного. Деструкторы выполняют противоположную операцию.

В результате этого в конструкторах и деструкторах виртуальные методы работают как невиртуальные.

В С++ множественное наследование подразумевает, что у одного класса может быть несколько базовых классов:

classTDelimitedReader : public TTextReader, public TStringList

{

...

};

Объект класса TDelimitedReader содержит все поля и методы базовых классов TTextReader иTStringList. При этом в классе TDelimitedReader можно переопределять виртуальные методыкаждого базового класса.

Множественное наследование имеет ряд проблем:

- отсутствие эффективной реализации (неэффективность скрыта от программиста);

- неоднозначность, возникающая из-за того, что в базовых классах могут быть одноименные поля, а также методы с одинаковой сигнатурой;

- повторяющийся базовый класс в иерархии классов.

Таким образом, множественное наследование таит следующую проблему: заранее неизвестно от каких классов программист захочет унаследовать свой класс. Однако при создании класса использовать виртуальное наследование неэффективно, если наследуются поля, так как доступ к полям всегда будет осуществляться через дополнительный указатель.Вывод: одинарное наследование в стиле Java, C++, Delphi допустимо только от классов,множественное — от интерфейсов. Иначе можно осуществлять множественное наследование лишь от классов, в которых отсутствуют поля.

**30)** В С++ виртуальные методы определяются при помощи ключевого слова virtual:

class TTextReader: virtual public TObject

{

...

};

При переопределении виртуального метода ключевое слово virtual можно записать, а можно опустить. Синтаксис переопределения виртуальных методов не предусматривает такие проблемы, как версионность и рефакторинг кода (упрощение программного кода с сохранением функциональности).

Если метод виртуальный следует всегда писать ключевое слово virtual.

В С++ абстрактный класс объявляется следующим образом:

class TTextReader

{

protected:

virtual void NextLine() = 0;

...

};

Такой метод называется абстрактным(или же чистой виртуальной функцией) и класс, содержащий данный метод, тоже называется абстрактным.

**31)**

Существует четыре оператора преобразования типа в С++:

reinterpret\_cast<тип>(переменная)

static\_cast<тип>(переменная)

const\_cast<тип>(переменная)

dynamic\_cast<тип>(переменная)

Первый оператор (reinterpret\_cast) позволяет отключить контроль типов данных на уровне компилятора, с помощью него любой указатель может быть интерпретирован, как любой другой указатель, а также любая память или переменная может быть интерпретирована иначе.В программах этот оператор преобразования типа использовать не следует, так как он нарушает переносимость программ. Его наличие свидетельствует о том, что программа является кросс-платформенной. Пример использования:

int i;

char \*p;

p = reinterpret\_cast<char>(&i);

Второй оператор (static\_cast) используется вместо преобразования тип(переменная), (тип)переменная и (тип)(переменная) при работе с классами, структурами и указателями на них. Оператор static\_cast был задуман по причине того, что в С++ выражение тип(переменная) может оказаться вызовом конструктора. Если в программе требуется преобразовать тип, а не вызвать конструктор типа, используется данный оператор. Кроме того, оператор (тип)переменная или (тип)(переменная) может в некоторых случаях оказаться преобразованием reinterpret\_cast<тип>(переменная), а при разработке кросс-платформенных программ оператор reinterpret\_cast всегда содержит потенциальную опасность неправильной работы программы на другой платформе. Поэтому вместо операторов тип(переменная), (тип)переменная и (тип)(переменная) следует использовать операторы reinterpret\_cast и static\_cast, которые убирают не явность из преобразования.

Так как оператор static\_cast является громоздким, то для простых типов данных допустимо использование форм: (тип)переменная и (тип)(переменная). Форма тип(переменная) не должна использоваться для преобразования типа.

Третий оператор (const\_cast) используется для приведения не константных указателей константным и наоборот. При объявлении переменных и параметров функций в описании типа может быть указано ключевое слово const. Объявление f(const char \*s); означает, что символы, адресуемые указателем s, изменять нельзя. Наличие константных объектов порождает проблему — огромная избыточность программного кода. Заранее программист не знает, будет ли пользователь (другой программист) его класса создавать константные объекты. Вследствие того, что это не исключено, программист начинает записывать слово const в объявление всех методов, в которых его можно записать. Многие методы являются виртуальными или вызывают виртуальные методы. Случается так, что в производных классах виртуальные методы, вызванные константными методами, модифицируют поля объектов (это требуется по условию задачи). Это приводит к логической проблеме, которая решается либо за счет применения оператора const\_cast к указателю this в производных классах, либо за счет объявления полей в производных классах с модификатором mutable (записывается при описании полей класса в том случае, если они должны модифицироваться константными методами).

**33)** Перегрузка операторов позволяет заменить смысл стандартных операторов (+, –, = и др.) для пользовательских типов данных. В С++ разрешена перегрузка операторов, выраженных в виде символов, а также операторов:

new delete

new[] delete[]

Запрещена перегрузка следующих операторов:

:: . .\* ?:

Перегрузка операторов таит угрозу: она резко усложняет понимание программы, поэтому ей пользоваться нужно очень осторожно. Для стандартных типов данных перегрузка запрещена, хотя бы один из операторов должен принадлежать пользовательскому типу данных.

Бинарный оператор можно определить либо в виде нестатической функции членов с одним аргументом, либо в виде статической функции с двумя аргументами.

Для любого бинарного оператора @ выражение aa@bb интерпретируется как

aa.operator@(bb) или operator@(aa, bb). Если определены оба варианта, то применяется

механизм разрешения перегрузки функций.

Унарные операторы бывают префиксными и постфиксными.

Унарный оператор можно определить в виде метода класса без аргументов и в виде функции с одним аргументом. Аргумент функции — объект некоторого класса.

Для любого префиксного унарного оператора выражение @aa интерпретируется как:

aa.operator @();

operator @(aa);

Для любого постфиксного унарного оператора выражение aa@ интерпретируется, как:

aa.operator @(int);

operator @(aa, int);

В С++ существуют операторы преобразования типов. Это является хорошим способом использования конструктора для преобразования типа. Конструктор не может выполнять следующие преобразования:

-неявное преобразование из типа, определяемого пользователем в базовый тип. Это связано с тем, что базовые типы не являются классами.

- преобразование из нового класса в ранее определенный класс, не модифицируя

объявление ранее определенного класса.

Оператор преобразования типа возвращает значение типа T, однако в сигнатуре оператора он не указывается. В этом смысле операторы преобразования типа похожи на конструкторы. Хотя конструктор не может использоваться для неявного преобразования типа из класса в базовый тип, он может использоваться для неявного преобразования типа из класса в класс. В программе следует избегать любых неявных преобразований типов, так как это приводит к ошибкам. С помощью ключевого слова explicit можно запретить неявное преобразования типа к конструкторам. Слово explicit записывается лишь для тех конструкторов, которые могут вызываться лишь с одним параметром. Если же они вызываются с несколькими параметрами, то неявное преобразование типов невозможно.

**34)** Шаблоны обеспечивают непосредственную поддержку обобщенного программирования. Они представляют собой параметризованные классы и параметризованные имена функций. Шаблон определяется с помощью ключевого слова template:

template <class T>

class basic\_string

{

public:

basic\_string();

basic\_string(const T\*);

basic\_string(const basic\_string&);

private:

T\*str;

};

typedef basic\_string<char> string;

typedef basic\_string<unsigned int> wstring;

Вместо слова typename часто записывают слово class, но параметром шаблона может быть любой тип данных. С точки зрения компилятора, шаблон является макроподстановкой, поэтому шаблонные классы определяются целиком в заголовках файлов (в h-файле, а не в cpp-файле).

Методы шаблона описываются следующим образом:

template <class T>

basic\_string<T>::basic\_string(const \*T)

{

...

}

Допускается применение шаблонов с целью реализации абстрактных алгоритмов, то есть шаблонов функций.

template <class T>

void sort(vector<T>& v);

При вызове шаблонных функций компилятор подставляет тип данных и создает новый

вариант функции. Если один и тот же тип данных используется несколько раз, то на все типы данных используется несколько раз, то на все типы данных создается один шаблон функции. При использовании шаблонов существует три больших недостатка:

-шаблоны невозможно отлаживать.

-существенно замедляется время компиляции. В больших проектах оно может

доходить до 30-60 минут.

- очень быстро растут размеры объектных модулей и библиотек на диске.

Шаблонные функции могут вызываться с явным указанием параметра шаблона:

sqrt<int>(2);

или без него:

sqrt(2);

В этом случае применяется механизм разрешения перегрузки:

- ищется набор специализации шаблонов функций, которые примут участие в

разрешении перегрузки;

- если могут быть вызваны два шаблона функций и один из них более специализирован,

то только он и будет рассматриваться;

- разрешается перегрузка для этого набора функций и любых обычных функций. Если

аргументы функции шаблона были определены путем выведения по фактическим

аргументам шаблона, к ним нельзя применять “продвижение” типа, стандартные и

определяемые пользователем преобразования.

- если и обычная функция, и специализация подходят одинаково хорошо, предпочтение

отдается обычной функции;

- если ни одного соответствия не найдено, или существует несколько одинаково хорошо подходящих вариантов, то выдается ошибка. В параметрах шаблонов допустимы стандартные значения, принимаемые по умолчанию.

Как правило, шаблон представляет единственное определение, которое применяется к различным аргументам шаблона. Это не всегда удобно, иногда существует необходимость использовать различные реализации в зависимости от типа. Например, надо для всех указателей использовать особую реализацию шаблона, а для всех базовых типов данных — обычную реализацию. Это делается с помощью специализации шаблона. Специализация шаблонов, как правило, используется для сокращения объема программного кода. Если шаблон создается для указателей на какие-то объекты и класс объекта не так важен, то при использовании обычных шаблонов без специализации возникает многократное дублирование одного и того же кода. Это связано с тем, что в машинных кодах работа со всеми указателями строится одинаково. Чтобы избежать дублирования кода в случае использования указателей следует создавать специализации шаблонов.

[**https://metanit.com/cpp/tutorial/9.3.php**](https://metanit.com/cpp/tutorial/9.3.php) **- про специализацию шаблона**

**35) На основании всего вышесказанного**