## 1. Объект как динамический модуль. Понятие модуля. Принципы модульного программирования. Разграничение доступа к элементам модуля. Термин «инкапсуляция». Понятие объекта как динамического модуля. Понятие класса как описания динамического модуля.

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

*Модульное программирование – это организация программы как совокупности небольших независимых блоков, называемых модулями, структура и поведение которых подчиняются определенным правилам.*

Использование модульного программирования позволяет упростить тестирование программы и обнаружение ошибок. Аппаратно-зависимые подзадачи могут быть строго отделены от других подзадач, что улучшает мобильность создаваемых программ.

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 модулям осуществляется с помощью ключевых слов.

- public - доступ получают все;

- protected – доступ получает данный модуль и модули – расширения(наследники);  
- internal – доступ получает данный модуль и программы, где данный модуль подключается на уровне исходного кода;

- protected internal – доступ получают модули расширения и программы, где данный модуль подключается на уровне исходного кода;

- private – доступ только у самого модуля.

Есть еще доступ по умолчанию (в Java условно называется Friendly, но само это слово не пишется перед именем классе (не friendly class Examle, а просто class Example)). Доступ имеют все модули внутри пакета. Это касается Java, как называется и работает в других языках не шарю.

*Объект – динамический модуль, у которого может быть много экземпляров. Описание динамического модуля – класс.*

*Инкапсуляция – механизм, объединяющий данные и код, манипулирующий этими данными, а также защищающий код и данные от внешнего вмешательства или неправильного использования.*

## 2. Исключение. Понятия ошибки и исключения. Создание исключений. Классы исключений. Обработка исключений. Защита ресурсов от утечки в случае исключений. Приемы надежного программирования.

*Исключение – событие и объект с информацией об ошибке. Исключение может создаваться:*

- *Аппаратно* – в результате ошибки выполнения оператора;

- *Программно* – с помощью оператора создания исключения (throw new …).

Некоторые стандартные исключения модуля System:

Exception

ArithmeticException

DivideByZeroException

OverflowException

ArrayTypeMismatchException

Обработка исключений:

Try {

} catch {

}

Частичная обработка исключений:

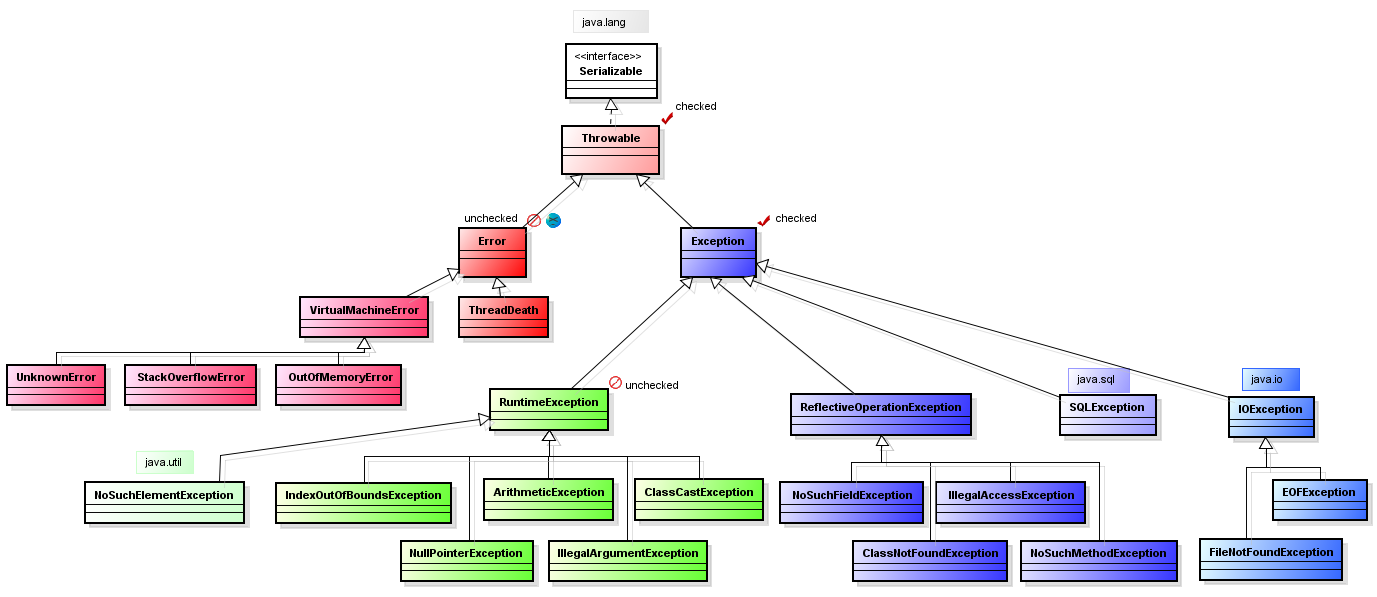
Try {

} catch {

Throw; //передача обработки внешнему блоку try .. catch

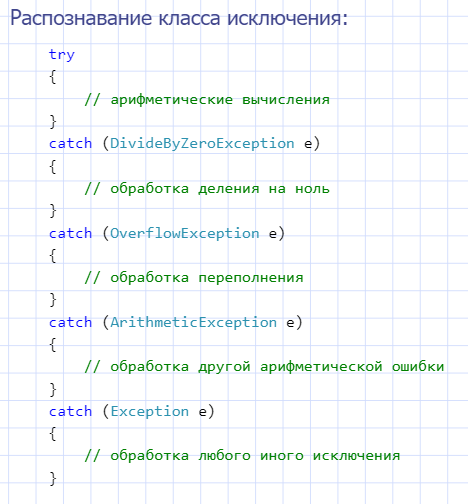
}

Классы исключений Java:



Проверяемые (checked) – те, где среда подсвечивает и пишет, что нужно обработать исключение.

Непроверяемые – где среда не дает подсказок.



Защита ресурсов от исключений выполняется с помощью finally – блок файнели будет выполнен в любом случае после блока try (не важно, произошло исключение или нет).

Try{

} finally {

}

В шарпах есть аналог блока try – finally :

Using (…){

}

В конце автоматически вызывается Dispose. Важно, что данная конструкция применяется только для классов, которые реализуют интерфейс IDisposable.

Комплексный вариант оператора try:

Try{

} catch (Exception e){

} finally {

}

## 3. Класс и объект. Понятие класса. Создание объекта. Понятие метода. Представление метода в виде обычной процедуры. Понятие конструктора и деструктора.

Класс – тип данных для создания объектов.

Объект – экземпляр класса.

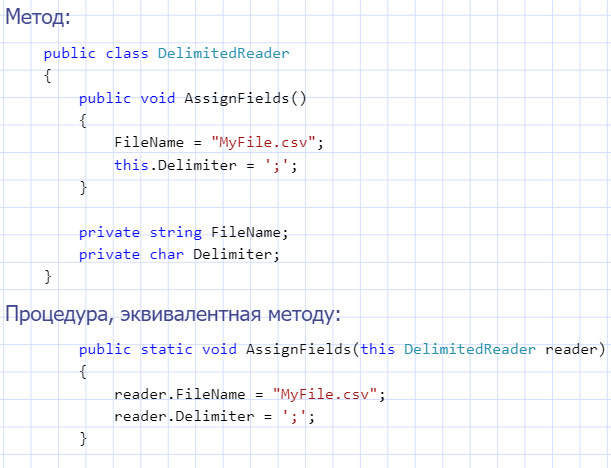
Метод – процедура над объектом.

Конструктор / деструктор – особые методы.

Объект создается словом New.

В метод передается скрытый параметр this (ссылка на объект, у которого вызван метод).

Представление метода в виде обычной процедуры: (просто явно передать параметр this)



Конструктор вызывается при создании нового объекта класса. Вызывать конструктор явно можно только из другого конструктора (их можно перегружать).

Деструктор вызывается для гарантированного освобождения памяти объекта. Вызывается при сборке мусора автоматически.

Синтаксис в Шарпах и С++:

~MyClass(){

}

В Джаве деструктора нет.

Функция - подпрограмма, выполняющая какие-либо операции и возвращающая значение.

Процедура - подпрограмма, которая только выполняет операции, без возврата значения.

Метод - это функция или процедура, которая принадлежит классу или экземпляру класса.

## 4. Свойство. Понятие свойства как виртуального поля. Назначение свойства. Методы получения и установки значений свойства. Режимы доступа к свойству. Свойство-индексатор или свойство-массив (в зависимости от языка программирования).

*Свойство – виртуальное поле (специальный метод доступа).*

*Свойство — предоставляют гибкий механизм для чтения, записи или вычисления значения частного поля.*

*Свойства позволяют классу предоставлять общий способ получения и задания значений, скрывая при этом код реализации или проверки.*

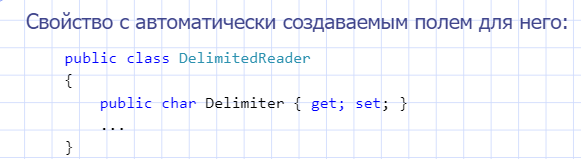
*Работа со свойством внешне не отличается от работы с полем.*

*Свойства выставляют поля. Поля должны (почти всегда) быть закрытыми для класса и доступны через свойства get и set. Свойства предоставляют уровень абстракции, позволяющий изменять поля, не влияя на внешний способ доступа к ним тем, что используют ваш класс.*

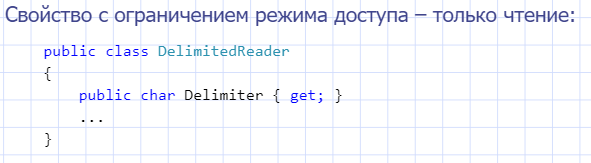
Стандартное определение свойства содержит блоки get и set. В блоке get мы возвращаем значение поля, а в блоке set устанавливаем. Параметр value представляет передаваемое значение. В блоках гет и сет можно писать любой код:



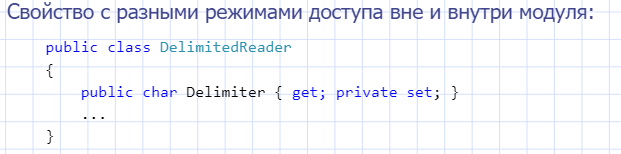
Свойство может быть с автоматически создаваемым для него полем (в примере сверху свойство Age задавало значение полю age):



Свойство может не содержать блока Get или Set (такое свойство называют «с режимом доступа только чтение / только запись»).



Свойство может изменять режим доступа:



Индексатор – особое свойство, предоставляющее доступ к объекту как к массиву.



## 5. Расширение класса. Расширение класса путем создания производного класса. Термин «наследование». Существование «прародителя» всех классов. Перекрытие элементов класса в производных классах. Совместимость объектов различных классов. Контроль и преобразование типов.

Наследование – процесс, посредством которого один объект может наследовать свойства другого объекта и добавлять к ним черты, характерные только для него.

Наследование бывает:

- одиночное (каждый класс всегда содержит 1ого предка);

- множественное (любое количество предков).

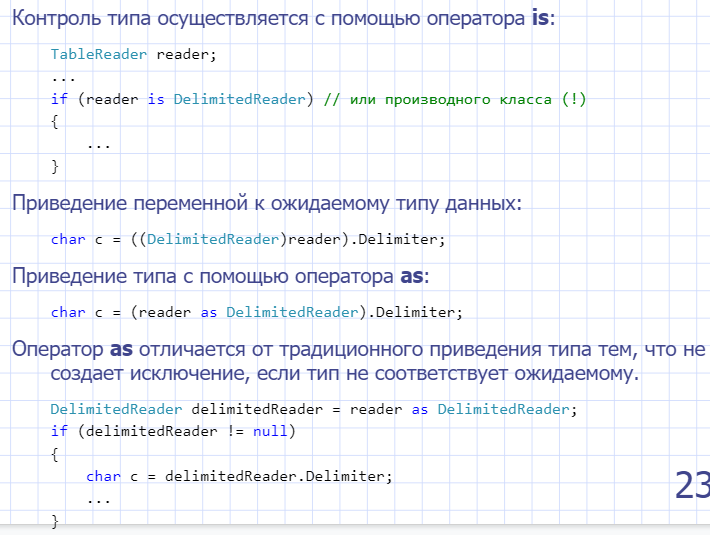
В шарпах и c++ расширяется через символ : , в джаве словом extends.

Public class myClass : myclass2;

Public class myClass extends myclass2;

Смысл расширения заключается в том, чтобы определить переменную базового типа и присваивать ей объекты любых производных типов.

Через переменную производного типа можно безопасно обращаться к полям и методам, определенным в базовом типе. Это обеспечивается благодаря бинарной совместимости всех производных типов с базовым.



В Java есть оператор instanceof, позволяющий проверить, является ли тип наследником другого типа.

Для всех классов в Java и C# базовый класс – Object, то есть все существующие классы наследуются от него. В С++ такого класса нет.

Методы класса Object в Java:

hashCode()

equals()

getClass()

clone()

finalize()

toString()

В С#:

GetType()

getHashCode()

ToString()

Equals()

## 6. Виртуальный метод и виртуальное свойство. Понятие виртуального метода. Перекрытие виртуального метода в производном классе. Абстрактный виртуальный метод. Механизм вызова виртуального метода. Динамический виртуальный метод (в некоторых языках программирования). Метод обработки сообщений (в некоторых языках программирования). Термин «полиморфизм». Понятие виртуального свойства. Запрет на расширение класса.

{Копипаст презентации}

Виртуальный метод – переопределяемый в производных классах метод. Вызов виртуального метода осуществляется в соответствии с фактическим типом объекта, к которому метод применяется.

Пример:

public class Object

{

...

public virtual string ToString();

}

public class Point : Object

{

public int X { get; protected set; }

public int Y { get; protected set; }

public Point(int x, int y) { X = x; Y = y; }

...

public override string ToString()

{

return String.Format("X: {0}, Y: {1}", X, Y);

}

}

...

object obj = new Point(10, 20);

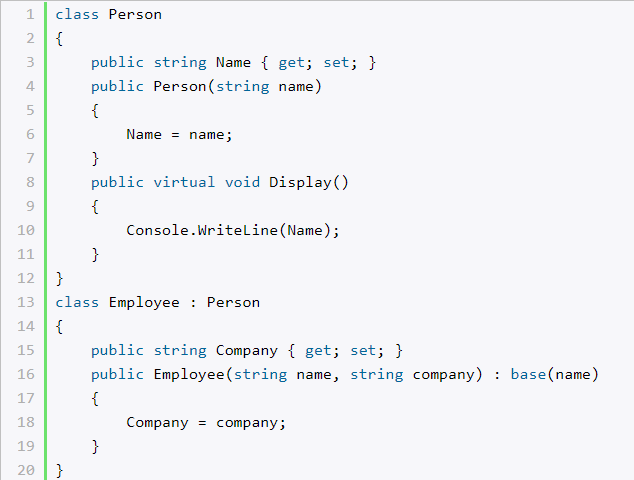
string str = obj.ToString(); // str = "X: 10, Y: 20"

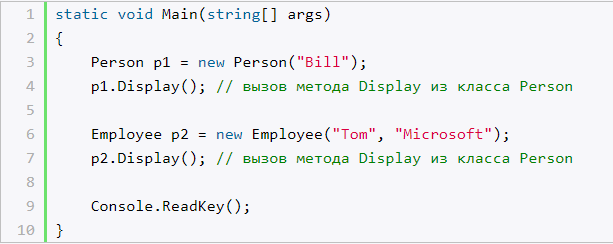
{<https://metanit.com/sharp/tutorial/3.19.php>}

{C#}

*virtual* – модификатор, которым обозначают те методы и свойства, которые нужно сделать доступными для переопределения. Чтобы переопределить метод в классе-наследнике, используется модификатор *override*.

Поля, помеченные *virtual*, переопределять не обязательно. В таком случае, будет вызываться реализация метода из класса-родителя.



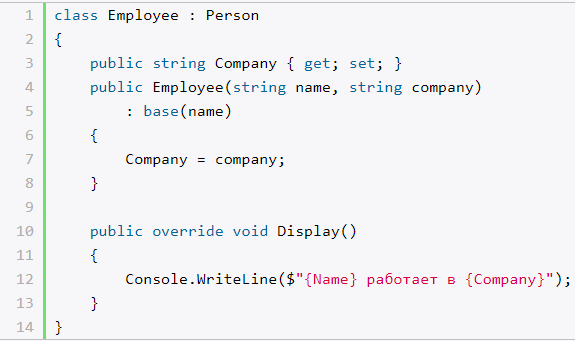


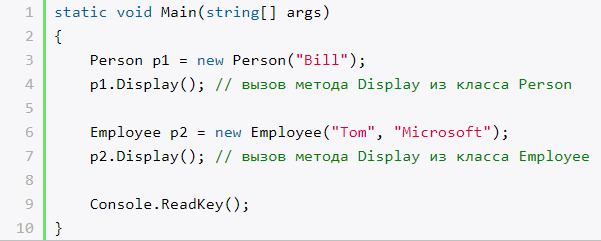
Консольный вывод:



Если же метод переопределить, то вывод будет другой. Для этого в классе-потомке нужно объявить новый метод с модификатором *override* с таким же названием, как и соответствующий виртуальный метод класса-родителя.

Переопределение виртуального свойства означает переопределение геттеров и сеттеров этого свойства. При этом, изменять их атрибуты нельзя.





Консольный вывод:



Также, переопределить виртуальные методы и свойства можно и в классах, не являющихся прямыми потомками базового класса. Т.е. если в примере выше классу Employee сделать потомка в виде класса Manager, то в нём тоже можно будет переопределить метод Display().

У виртуальных методов и свойств есть несколько ограничений:

1) У виртуальных и переопределённых методов и свойств должны быть одинаковые модификаторы доступа (виртуальный public -> переопределённый тоже должен быть public)

2) Статические методы и свойства нельзя ни переопределить, ни объявить виртуальными

С помощью ключевого слова *base* в С# можно обращаться к полям класса-родителя. Т.е. если в переопределённом методе класса Employee прописать

base.Display()

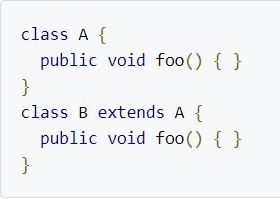
вызовется метод Display() из класса Person.

Запретить переопределение методов можно с помощью модификатора *sealed*, но он может использоваться только в связке с модификатором *override*. После этого, если сделать метод Display() в классе Employee *sealed*, то в классе Manager его нельзя будет переопределить. При этом, с этим модификатором можно объявить классы, что запретит их расширение.

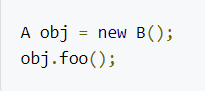
{Java}

В Java виртуальными полями считаются все поля, которые можно переопределить в классах-потомках. *Final* методы в Java переопределить нельзя (и расширить классы с таким модификатором тоже нельзя соответственно), и такие поля уже считаться виртуальными не будут. Поля интерфейсов и абстрактных классов также являются виртуальными, так как изначально спроектированы быть переопределёнными. При этом, явных модификаторов для виртуальных полей в этом ЯП нет.

Т.е., пример (псевдокод):



Если создать



То вызовется метод foo() класса B.

По этому для Java в таких случаях лучше использовать термин «Динамическая привязка».

По факту, JVM (как и виртуальная машина C#) ищет последнюю реализацию объявленного поля, и вызовет именно его.

{Общая инфа}

Абстрактный метод – виртуальный метод, реализация для которого отсутствует. Экземпляры классов, где есть хотя бы один не переопределённый абстрактный метод, создавать нельзя.

Механизм вызова виртуального метода:

1. Через объектную переменную выполняется обращение к данным объекта в динамической памяти (объектная переменная – ссылка на объект в памяти) (MOV EAX, obj)

2. Из этих данных извлекается адрес таблицы виртуальных методов (первые 4 или 8 байт, что зависит от разрядности ЦП) (MOV EBX, [EAX+0])

3. Извлекаем адрес вызываемого метода (MOV EAX, [EBX+8], где 8 – смещение до нужного нам поля таблицы виртуальных методо)

4. Делаем вызов метода (CALL EAX)

Динамический метод – метод, который вызвали динамически. Грубо говоря, вызов метода, о котором что-то известно будет уже во время исполнения программы (в runtime). (выделил жёлтым, т.к. не уверен в такой формулировке) (Сурков ещё пишет, что динамические методы не поддерживаются в Шарпах, но на деле же можно применить рефлексию, например, или словарь из делегатов (в Джаве словарь = Map)).

Динамические методы отличаются от виртуальных в механизме вызова. Первые экономят память, но медленные, а вторые быстрые, но тратят много памяти (таблицы динамических методов имеют более компактный вид).

Из возможности переопределения методов и свойств, вытекает понятие полиморфизма:

Полиморфизм – возможность объектов с одинаковой спецификацией иметь различную реализацию (т.е. Полиморфизм – это свойство системы использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта).

## 7. Делегат. Понятие делегата (ссылки на метод объекта – в зависимости от языка программирования). Пример применения делегата. Понятие события как списка делегатов. Пример применения события. Методы регистрации события.

{C#}

Делегат это просто механизм который позволяет делать ссылку на метод как тип данных, делегаты это синтаксический сахар, они позволяют делать код короче.

Делегат – ссылка на метод.

(другое определение)

Делегат – объект, который указывает на методы.

Делегат – указатель на метод.

Объявляется с помощью ключевого слова *delegate*.

class Program

{

    delegate void Message(); // 1. Объявляем делегат

    static void Main(string[] args)

    {

        Message mes; // 2. Создаем переменную делегата

        if (DateTime.Now.Hour < 12)

        {

            mes = GoodMorning; // 3. Присваиваем этой переменной адрес метода

        }

        else

        {

            mes = GoodEvening;

        }

        mes(); // 4. Вызываем метод

        Console.ReadKey();

    }

    private static void GoodMorning()

    {

        Console.WriteLine("Good Morning");

    }

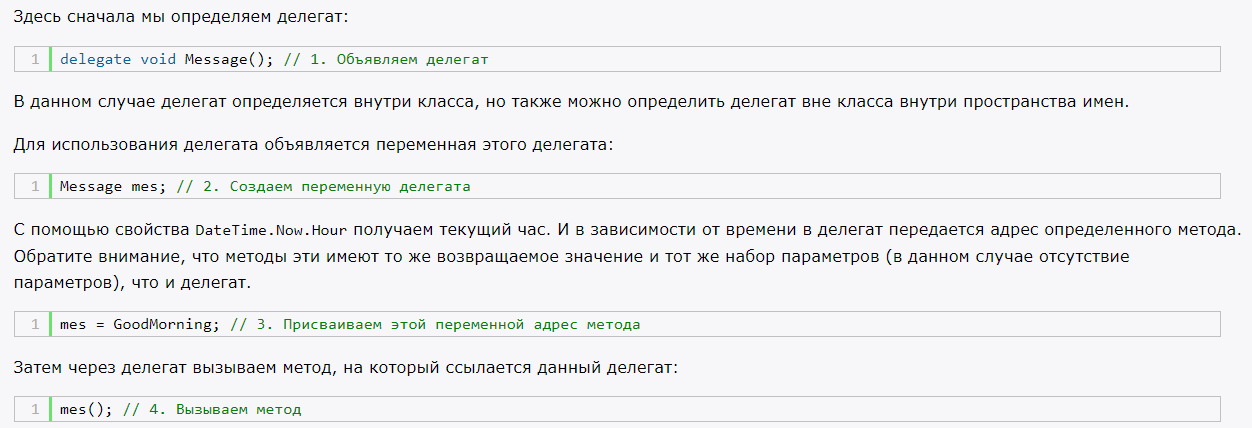
    private static void GoodEvening()

    {

        Console.WriteLine("Good Evening");

    }

}



{Java}

В Java нет делегатов. Ссылки на методы объекта можно объявить через лямбда-выражения.

Лямбда-выражения в Java выполняются как реализация метода в функциональном интерфейсе, в котором будет всего один метод без реализации. Т.е.:

public class LambdaApp {

    public static void main(String[] args) {

        Operationable operation;

        operation = (x,y)->x+y; //лямбда-выражение

        int result = operation.calculate(10, 20);

        System.out.println(result); //30

    }

}

interface Operationable{

    int calculate(int x, int y);

}



{C#} (<https://metanit.com/sharp/tutorial/3.14.php>)

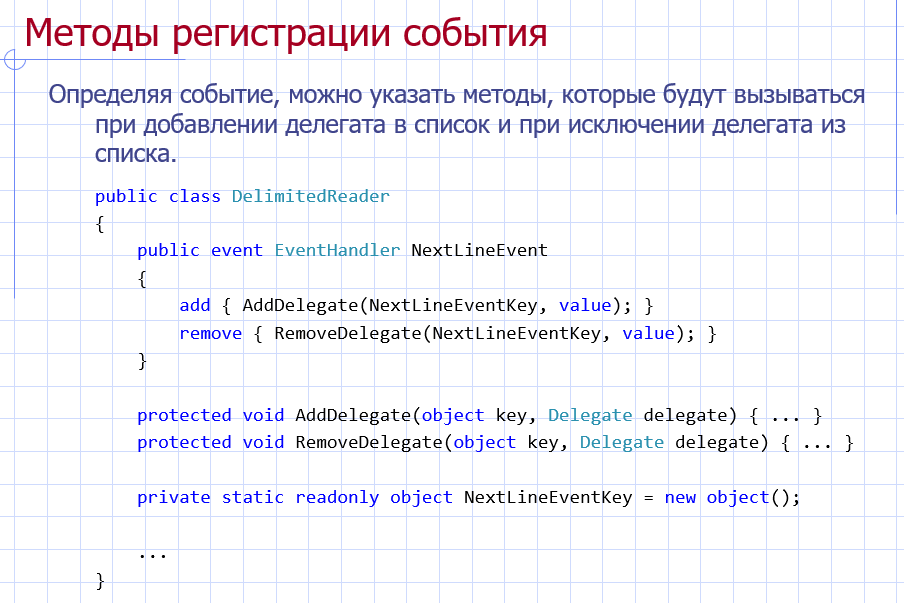
Событием называется список делегатов. Мы должны создать переменную-событие через ключевое слово *event*.

public event NextLineDelegate NextLineEvent;

Ну и через += добавляем к этой переменной методы.

(Как можем и убирать через -=)

Вызывать событие мы сможем через вот эту же событийную переменную.



## 8. Метакласс. Понятие метакласса (в некоторых языках программирования). Методы, применяемые к классам. Виртуальные конструкторы (в некоторых языках).

Метакласс – специальный класс для описания других классов.

В Шарпах отсутствуют.

В Джаве есть класс Class, который в рефлексии используется (<https://overcoder.net/q/476663/%D1%80%D0%B5%D0%B0%D0%BB%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F-java-%D0%BC%D0%B5%D1%82%D0%B0-%D0%BA%D0%BB%D0%B0%D1%81%D1%81%D1%8B>)

Вообще вопрос странный пизда, я не уверен, что тут писать можно, но эта штука есть в, внимание, ДЕЛФИ, поэтому напишу сюда про него, учитывая, что в шарпах его нет, а Джава тут нужна тока мне и Максу.

Язык Delphi позволяет рассматривать классы объектов как своего рода объекты, которыми можно манипулировать в программе. Такая возможность рождает новое понятие — класс класса; его принято обозначать термином метакласс. Для поддержки метаклассов введен специальный тип данных — ссылка на класс (classreference). Он описывается с помощью словосочетания classof. По аналогии с тем, как для всех классов существует общий предок TObject, у ссылок на классы существует базовый тип TClass, определенный, как:

type

TClass = class of TObject;

Переменная типа TClass может ссылаться на любой класс. Практическая ценность ссылок на классы состоит в возможности создавать программные модули, работающие с любыми классами объектов, даже теми, которые еще не разработаны.

Метаклассы привели к возникновению нового типа методов — методов класса. Метод класса оперирует не экземпляром объекта, а непосредственно классом. Он объявляется как обычный метод, но перед словом procedure или function записывается зарезервированное слово class.

Методы классов могут быть виртуальными. Например, в классе TObject определен виртуальный метод класса NewInstance. Он служит для распределения памяти под объект и автоматически вызывается конструктором. Его можно перекрыть в своем классе, чтобы обеспечить нестандартный способ выделения памяти для экземпляров. Метод NewInstance должен перекрываться вместе с другим методом FreeInstance, который автоматически вызывается из деструктора и служит для освобождения памяти. Добавим, что размер памяти, требуемый для экземпляра, можно узнать вызовом предопределенного метода класса InstanceSize.

Особые возможности ссылок на классы проявляется в сочетании с виртуальными конструкторами. Виртуальный конструктор объявляется с ключевым словом virtual. Вызов виртуального конструктора происходит по фактическому значению ссылки на класс, а не по ее формальному типу. Это позволяет создавать объекты, классы которых неизвестны на этапе компиляции. Механизм виртуальных конструкторов применяется в среде Delphi при восстановлении компонентов формы из файла. Восстановление компонента происходит следующим образом. Из файла считывается имя класса. По этому имени отыскивается ссылка на класс (метакласс). У метакласса вызывается виртуальный конструктор, который создает объект нужного класса.

## 9. Интерфейс. Понятие интерфейса. Описание интерфейса. Поддержка интерфейса классом. Механизм подсчета ссылок в интерфейсах. Расширение интерфейса. Глобально-уникальный идентификатор интерфейса. Совместимость интерфейсов и классов. Получение интерфейса через другой интерфейс. Представление интерфейса в памяти. Механизм вызова метода объекта через интерфейс.

Интерфейс = объект – реализация.

Интерфейс – это набор методов класса, доступных для использования другими классами.

Для определения интерфейса используется ключевое слово *interface*. Как правило, названия интерфейсов в C# начинаются с заглавной буквы I, например, IComparable, IEnumerable (так называемая венгерская нотация), однако это не обязательное требование, а больше стиль программирования.

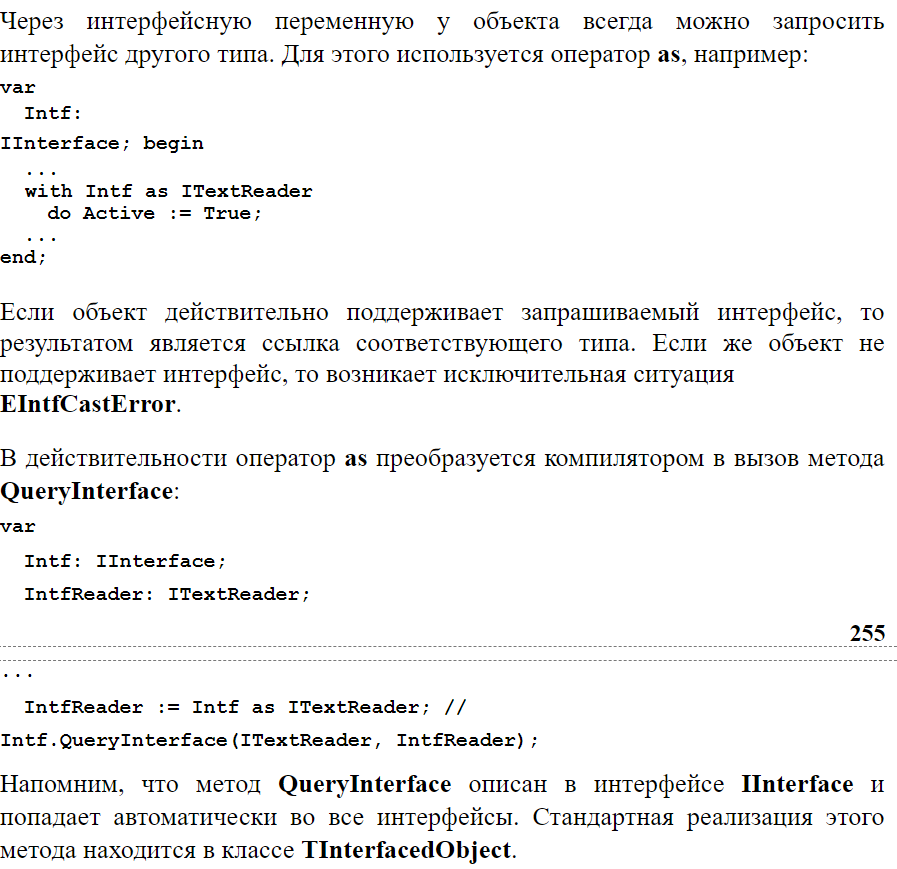
Интерфейс не может содержать поля, конструкторы, деструктор. Все элементы интерфейса по определению являются общедоступными (public) и абстрактными.



Начиная с версии C# 8.0 интерфейсы поддерживают реализацию методов и свойств по умолчанию. Это значит, что мы можем определить в интерфейсах полноценные методы и свойства, которые имеют реализацию как в обычных классах и структурах.

С реализацией свойств по умолчанию в интерфейсах дело обстоит несколько сложнее, поскольку мы не можем определять в интерфейсах нестатические переменные, соответственно в свойствах интерфейса мы не можем манипулировать состоянием объекта. Тем не менее реализацию по умолчанию для свойств мы тоже можем определять. Стоит отметить, что если интерфейс имеет приватные методы и свойства (то есть с модификатором private), то они должны иметь реализацию по умолчанию. То же самое относится к любым статическим методам и свойствам (не обязательно приватным).

(я не совсем понимаю чё имелось в виду под получением интерфейса через интерфейс, поэтому ебать держите опять про Делфи)



(пока искал инфу для этого вопроса, нашёл где Сурков пиздил инфу <https://studfile.net/preview/5532718/page:68/#213>)

## 10. Шаблон. Понятие шаблона как параметризованного класса. Создание переменной на базе шаблонного типа. Представление данных, заданных шаблоном, в памяти. Параметры шаблона. Установка ограничений на параметры шаблона. Шаблон делегата.

Шаблон – класс, параметризованный типом данных.

Пример шаблона:

public class List<T>

{

public List();

public void Add(T item);

public void AddRange(List<T> collection);

public bool Remove(T item);

public T[] ToArray();

...

}

На основе шаблона можно сконструировать тип и создать переменную сконструированного типа:

var stringList = new List<string>();

var integerList = new List<int>();

Представление в памяти сконструированного типа зависит от типа-параметра. Представление оптимизировано для каждого отдельного скалярного типа-параметра (например, int, double и т.д.) и для всех ссылочных типов (например, object, string и т.д.).

При описании шаблона можно ограничить возможные значения параметра-типа:

public class List<T>

where T: ListItem<T>

{

...

public void Add(T item); // Внутри шаблона - List<T>.Add()

...

}

Вариант шаблона с несколькими параметрами и ограничениями:

public class Dictionary<K, V>

where K: IComparable<K>, ISerializable

where V: ISerializable

{

...

public void Add(K key, V value); // Dictionary<K, V>.Add()

...

}

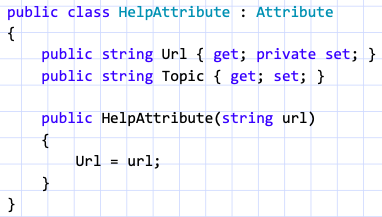
Пример шаблона делегата (LINQ) (<https://docs.microsoft.com/ru-ru/dotnet/csharp/delegates-patterns>):

public static IEnumerable<TSource> Where<TSource> (this IEnumerable<TSource> source, Func<TSource, bool> predicate);

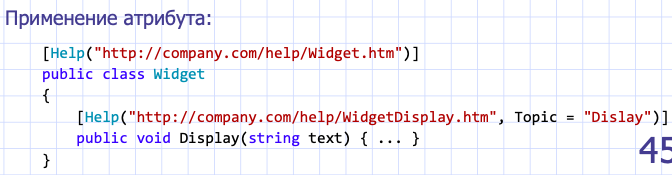
var smallNumbers = numbers.Where(n => n < 10);

## 11. Атрибут. Информация о классах во время выполнения программы. Понятие «рефлексии» объектов. Понятие атрибута (аннотации – в другом языке программирования) как механизма рефлексии. Создание пользовательского атрибута. Проверка атрибута во время выполнения программы. Пример применения атрибутов.

Атрибут – метаданные , которые можно назначать элементам программы. Эти метаданные представляются как объекты, производные от класса Attribute.



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

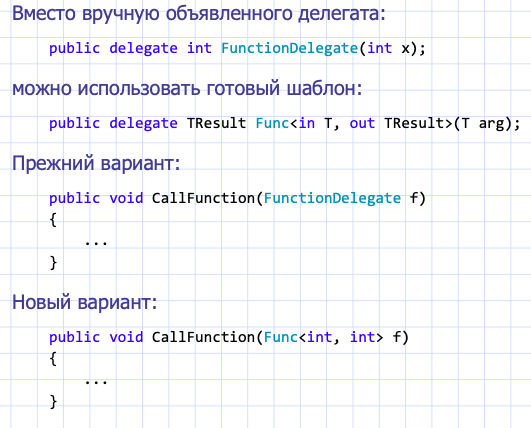


Атрибуты имеют следующие свойства.

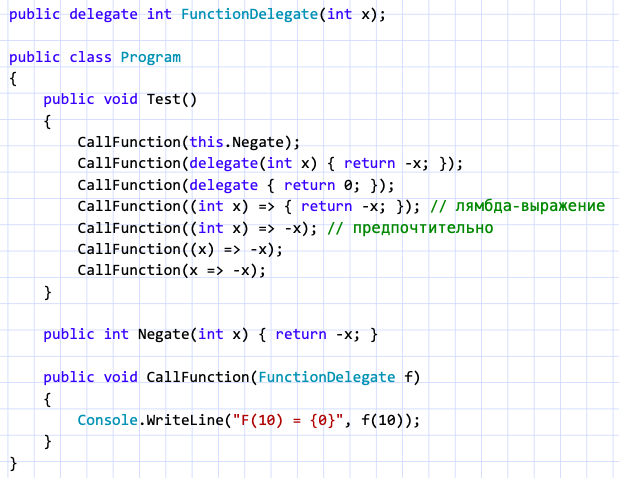
* Атрибуты добавляют в программу метаданные. Метаданные — это сведения о типах, определенных в программе. Все сборки .NET содержат некоторый набор метаданных, описывающих типы и члены типов, определенные в этой сборке. Вы можете добавить пользовательские атрибуты, чтобы указать любую дополнительную информацию. Подробнее см. в разделе Создание настраиваемых атрибутов (C#).
* Вы можете применить один или несколько атрибутов ко всей сборке, к модулю или к более мелким элементам программы, например к классам и свойствам.
* Атрибуты могут принимать аргументы, так же как методы и свойства.
* Программа может проверить собственные метаданные или метаданные в других программах, используя отражение. Дополнительные сведения см. в разделе Обращение к атрибутам с помощью отражения (C#).

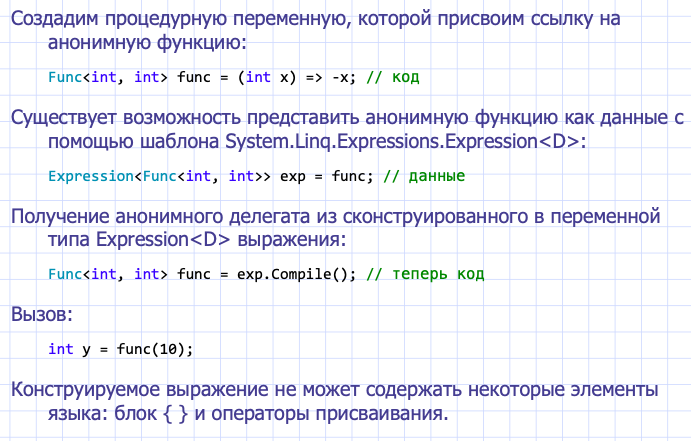
## 12. Анонимная функция. Варианты анонимной функции. Шаблон делегата-функции. Представление анонимной функции в виде данных. Ограничения на выражения внутри анонимной функции.

Анонимная функция — особый вид функций, которые объявляются в месте использования и не получают уникального идентификатора для доступа к ним. Поддерживаются во многих языках программирования. Шаблон:



Обычно при создании анонимные функции либо вызываются напрямую, либо ссылка на функцию присваивается переменной, с помощью которой затем можно косвенно вызывать данную функцию. Но в последнем случае анонимная функция получает имя и уже перестаёт быть анонимной. Если анонимная функция ссылается на переменные, не содержащиеся в её теле (захват), то такая функция называется замыканием. Лямбда-выражение — типичная для многих языков синтаксическая конструкция для определения анонимной функции.

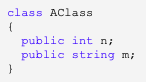


Представление в виде данных и ограничения:

## 13. Деление типов данных на типы-«значения» (value-types) и типы-«ссылки» (reference-types). Отличие структур (записей) от классов на платформе .NET. Автоматическое управление памятью ссылочных данных. Упаковка и разупаковка данных. Типы данных со значением null.

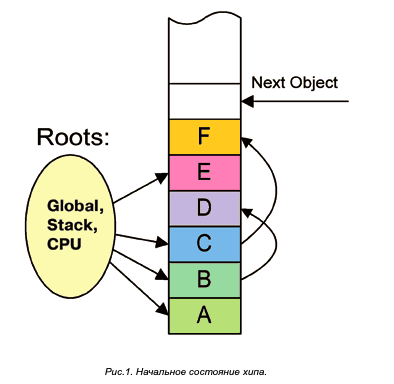
В среде .Net есть две категории типов – ссылочные типы и типы значения. То, к какой категории будет принадлежать определяемый тип, задаётся при его определении.

**Ссылочный тип** состоит как минимум из указателя на объект, когда он равен null, и также из динамического хранилища, на которое ссылается этот указатель. Таким образом, рабочий ссылочный тип использует 2 области памяти - место для хранения указателя (это стек) и место, где непосредственно содержатся данные объекта (это куча), на который ссылается указатель. Ссылочные типы в языке C# определяются с помощью ключевого слова class. ссылочные типы – это типы, унаследованные от любых типов, кроме типов System.ValueType и System.Enum. Тип Enum сам является наследником System.ValueType. Потому ссылочные типы – это все типы, кроме прямых или косвенных потомков System.ValueType.

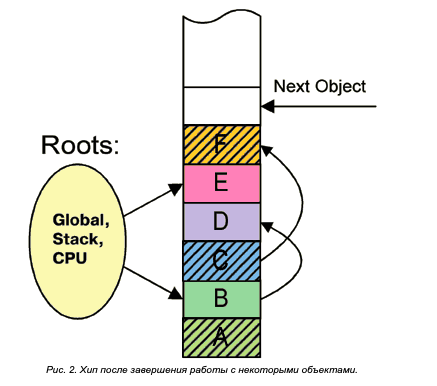


**Типы значений** - это такие типы переменной, которые хранят данные о себе только в одном месте. Это типы, определённые с помощью ключевых слов struct, enum, а также все фундаментальные (то есть, входящие в список основных предопределенных типов, непосредственно понимаемых компилятором) типы (int, char, float,…), за исключением типа System.String (System.String – это единственный ссылочный фундаментальный тип).

Автоматическое управление памятью ссылочных данных. Для хранения объектов CLR использует хип, подобный хипу C++, за тем важным исключением, что хип CLR не фрагментирован. Выделение объектов производится всегда один за другим в последовательных адресах памяти, что позволяет весьма существенно повысить производительность всего приложения



Когда память заканчивается, в дело вступает процесс, который мы опишем ниже. Фактически, основная задача этого процесса сводится к тому, чтобы освободить место в хипе путём дефрагментации неиспользуемой памяти.



Первое, что делает сборщик мусора во время работы – это принимает решение о том, что все выделенные блоки памяти в вашей программе - это как раз и есть мусор, и они вам больше не нужны. Но на этом сборщик мусора не прекращает свою работу. Далее начинается утомительный поиск «живых» указателей на объекты по всем закоулкам приложения. Microsoft называет эти указатели «roots». В процессе поиска сборщик мусора сканирует глобальную память программы (на рисунках обозначено как Global), стек (Stack – локальные переменные) и даже регистры процессора (CPU). Как мы знаем, каждый поток в программе имеет свой собственный стек, и CLR приходится сканировать их все. Кроме того, интересен тот факт, что CLR умеет работать даже с потоками, которые создавались в неуправляемом коде с использованием функций WinAPI. По крайней мере, следующий тест на MC++ отработал правильно, выдав на консоль результат – «212».

**Упаковка** представляет собой процесс преобразования типа значения в тип object или в любой другой тип интерфейса, реализуемый этим типом значения. Когда тип значения упаковывается общеязыковой средой выполнения (CLR), он инкапсулирует значение внутри экземпляра System.Object и сохраняет его в управляемой куче. Операция распаковки извлекает тип значения из объекта. Упаковка является неявной; распаковка является явной. Понятия упаковки и распаковки лежат в основе единой системы типов C#, в которой значение любого типа можно рассматривать как объект.

Пример процесса упаковки:

int i = 123;

// The following line boxes i.

object o = i;

Пример процесса распаковки:

o = 123;

i = (int)o; // unboxing

**Ссылка не должна иметь значение NULL.**

Когда переменные не должны иметь значение NULL, компилятор принудительно применяет правила, которые гарантируют, что можно разыменовать эти переменные без предварительной проверки того, что они не имеют значение NULL:

* Переменную необходимо инициализировать со значением, отличным от NULL.
* Переменной не может быть присвоено значение null.

Ссылка может иметь значение NULL. Когда переменные могут иметь значение NULL, компилятор применяет различные правила, чтобы убедиться, что вы правильно проверили наличие пустой ссылки:

* Переменная может быть разыменована только в том случае, когда компилятор может гарантировать, что значение не равно NULL.
* Эти переменные могут быть инициализированы со значением по умолчанию null, и им можно присвоить значение null в другом коде.

Пример, где допускается значение NULL:

string? name;

## 14. Объектно-ориентированное программирование в языке Java. Модуль, класс, объект, метод, атрибуты доступа к элементам объекта, интерфейс, шаблон, аннотация. Отличия средств объектно-ориентированного программирования в языках C# и Java.

**Класс** в Java - это шаблон для создания объекта, а **объект** - это экземпляр класса. Класс определяет структуру и поведение, которые будут совместно использоваться набором объектов. Класс содержит переменные и методы, которые называются элементами класса, членами класса. Он составляет основу инкапсуляции в Java. Каждый объект данного класса содержит структуру и поведение, которые определены классом. Иногда объекты называют экземплярами класса.

**Методы** используются для описания того, что объект класса умеет делать или что можно с ним сделать. Переменные - для описания свойств или характеристик объекта.

**Модуль** — это независимый блок развертывания (допускающий переиспользование в любом контексте). Модуль обладает стабильными конкретными идентификационными признаками (например, именем и версией). Модуль предъявляет определенные требования (например, зависимости). Один модуль может использоваться другими, но при этом внутренние детали данного модуля остаются для них скрытыми.

**Интерфейс** — это ссылочный тип в Java. Он схож с классом. Это совокупность абстрактных методов. Класс реализует интерфейс, таким образом наследуя абстрактные методы интерфейса.

Вместе с абстрактными методами интерфейс в Java может содержать константы, обычные методы, статические методы и вложенные типы. Тела методов существуют только для обычных методов и статических методов.

**Паттерны проектирования** (шаблоны проектирования) - это готовые к использованию решения часто возникающих в программировании задач. Паттерны проектирования, подходящий под задачу, реализуется в каждом конкретном случае.

Типы паттернов:

* Порождающие (Factory, Builder)
* структурные (Proxy, Bridge)
* поведенческие (State, Visitor)

**Аннотации** представляют из себя дескрипторы, включаемые в текст программы, и используются для хранения метаданных программного кода, необходимых на разных этапах жизненного цикла программы.

Информация, хранимая в аннотациях, может использоваться соответствующими обработчиками для создания необходимых вспомогательных файлов или для маркировки классов, полей и т.д.

**Атрибуты доступа:**

*public*: публичный, общедоступный класс или член класса. Поля и методы, объявленные с модификатором public, видны другим классам из текущего пакета и из внешних пакетов.

*private*: закрытый класс или член класса, противоположность модификатору public. Закрытый класс или член класса доступен только из кода в том же классе.

*protected*: такой класс или член класса доступен из любого места в текущем классе или пакете или в производных классах, даже если они находятся в других пакетах

*Модификатор по умолчанию*. Отсутствие модификатора у поля или метода класса предполагает применение к нему модификатора по умолчанию. Такие поля или методы видны всем классам в текущем пакете.

**Отличия средств ООП в Java от C#:**

* Все методы виртуальны.
* Также в java нет множественного наследования.
* Нет value-типов.
* Для обработки событий используются анонимные классы.
* Для NULL примитивов используются классы-обёртки(Integer, Long, Double).

## 15. Визуальное компонентное и сборочное программирование. Понятие компонента. Понятие визуального программирования. Инструментальные средства визуального компонентного программирования. Современные библиотеки компонентов. Понятие расширяемой программы и расширяемого программирования. Принципы расширяемого программирования. Применение методологии объектно-ориентированного программирования для построения расширяемых программ.

**Сборочное программирование**– вид подходов, предполагающих сборку программы из уже разработанных фрагментов. Сборка может осуществляться вручную, указываться на языке сборки или извлечена полуавтоматическим образом из спецификации. Фрагментами кода обычно выступают программные модули или иные части программы, представляемые в виде модулей определённого вида.

Выделяют следующие основные способы поддержки этого подхода:

1. Выработка стиля программирования, соответствующего принятым принципам модульности.

2. Повышение эффективности межмодульных интерфейсов. Обеспечение поддержки модульности на уровне программно-аппаратной платформы.

3. Ведение большой базы программных модулей. Решение проблемы идентификации модулей и проверки пригодности по описанию интерфейса.

**Визуальное программирование** предусматривает создание приложений с помощью наглядных средств. В процессе создания программы, программист показывает, что должно получится в результате, а код программы создается в полуавтоматическом режиме. В программировании можно визуализировать различные процессы: построение интерфейса, создание структуры БД, работа с сетью, программирование логики и т.д.

**Компонент** – это программный модуль или объект, который готов для использования в качестве составного блока программы и которым можно визуально манипулировать во время разработки программ.

**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, добавляют также новые (расширенные) типы данных.

**Объектно-ориентированное сборочное программирование.** Подход базируется на методологии объектно-ориентированного программирования и предполагает распространение библиотек классов в виде исходного кода или упаковку классов в динамически компонуемую библиотеку.

## 16. Принципы расширяемого программирования, известные как SOLID (Single responsibility, Open/closed, Liskov substitution, Interface segregation, Dependency inversion) и SOA (Service-Oriented Architecture).

**S - Принцип единственной обязанности** (Single Responsibility Principle) можно сформулировать так:

**У класса должна быть только одна причина для изменения**

Под обязанностью здесь понимается набор функций, которые выполняют единую задачу. Суть этого принципа заключается в том, что класс должен выполнять одну единственную задачу. Весь функционал класса должен быть целостным, обладать высокой связностью (high cohesion).

Конкретное применение принципа зависит от контекста. В данном случае важно понимать, как изменяется класс. Если класс выполняет несколько различных функций, и они изменяются по отдельности, то это как раз тот случай, когда можно применить принцип единственной обязанности. То есть иными словами, у класса несколько причин для изменения.

Но если же все функции класса, как правило, изменяются вместе и составляют одно функциональное целое, решают одну задачу, то нет смысла применять данный принцип.

**Распространенные случаи нарушения принципа SRP**

Нередко принцип единственной обязанности нарушает при смешивании в одном классе функциональности разных уровней. Например, класс производит вычисления и выводит их пользователю, то есть соединяет в себя бизнес-логику и работу с пользовательским интерфейсом. Либо класс управляет сохранением/получением данных и выполнением над ними вычислений, что также нежелательно. Класс следует применять только для одной задачи - либо бизнес-логика, либо вычисления, либо работа с данными.

Другой распространенный случай - наличие в классе или его методах абсолютно несвязанного между собой функционала.

**O - Принцип открытости/закрытости** (Open/Closed Principle) можно сформулировать так:

**Сущности программы должны быть открыты для расширения, но закрыты для изменения**.

Суть этого принципа состоит в том, что система должна быть построена таким образом, что все ее последующие изменения должны быть реализованы с помощью добавления нового кода, а не изменения уже существующего.

В первую очередь надо вынести из класса и инкапсулировать всю ту часть, которая представляет изменяющееся поведение. Однако это не всегда бывает просто сделать. Возможно, в классе много методов, но на начальном этапе сложно определить, какие из них будут изменять свое поведение и как изменять. В этом случае, конечно, надо анализировать возможные способы изменения и уже на основании анализа делать выводы. То есть, **все, что подается изменению, выносится из класса и инкапсулируется во вне - во внешних сущностях.**

**L - Принцип подстановки Лисков** (Liskov Substitution Principle) представляет собой некоторое руководство по созданию иерархий наследования. Изначальное определение данного принципа, которое было дано Барбарой Лисков в 1988 году, выглядело следующим образом:

Если для каждого объекта o1 типа S существует объект o2 типа T, такой, что для любой программы P, определенной в терминах T, поведение P не изменяется при замене o2 на o1, то S является подтипом T.

То есть иными словами класс S может считаться подклассом T, если замена объектов T на объекты S не приведет к изменению работы программы.

В общем случае данный принцип можно сформулировать так:

**Должна быть возможность вместо базового типа подставить любой его подтип**.

Фактически принцип подстановки Лисков помогает четче сформулировать иерархию классов, определить функционал для базовых и производных классов и избежать возможных проблем при применении полиморфизма.

Существует несколько типов правил, которые должны быть соблюдены для выполнения принципа подстановки Лисков. Прежде всего это правила контракта.

Контракт представляет собой некоторый интерфейс базового класса, некоторые соглашения по его использованию, которым должен следовать класс-наследник. Контракт задает ряд ограничений или правил, и производный класс должен выполнять эти правила:

* Предусловия (Preconditions) не могут быть усилены в подклассе. Другими словами подклассы не должны создавать больше предусловий, чем это определено в базовом классе, для выполнения некоторого поведения

Предусловия представляют набор условий, необходимых для безошибочного выполнения метода.

Причем объектом предусловий могут быть только общедоступные свойства или поля класса, или параметры метода. Приватное поле не может быть объектом для предусловия, так как оно не может быть установлено из вызывающего кода.

* Постусловия (Postconditions) не могут быть ослаблены в подклассе. То есть подклассы должны выполнять все постусловия, которые определены в базовом классе.

Постусловия проверяют состояние возвращаемого объекта на выходе из функции.

* Инварианты (Invariants) — все условия базового класса - также должны быть сохранены и в подклассе

Инварианты - это некоторые условия, которые остаются истинными на протяжении всей жизни объекта. Как правило, инварианты передают внутреннее состояние объекта.

**I - Принцип разделения интерфейсов** (Interface Segregation Principle) относится к тем случаям, когда классы имеют "жирный интерфейс", то есть слишком раздутый интерфейс, не все методы и свойства которого используются и могут быть востребованы. Таким образом, интерфейс получатся слишком избыточен или "жирным".

Принцип разделения интерфейсов можно сформулировать так:

**Клиенты не должны вынужденно зависеть от методов, которыми не пользуются.**

При нарушении этого принципа клиент, использующий некоторый интерфейс со всеми его методами, зависит от методов, которыми не пользуется, и поэтому оказывается восприимчив к изменениям в этих методах. В итоге мы приходим к жесткой зависимости между различными частями интерфейса, которые могут быть не связаны при его реализации.

В этом случае интерфейс класса разделяется на отдельные части, которые составляют раздельные интерфейсы. Затем эти интерфейсы независимо друг от друга могут применяться и изменяться. В итоге применение принципа разделения интерфейсов делает систему слабосвязанной, и тем самым ее легче модифицировать и обновлять.

interface ICall

{

void Call();

}

interface IPhoto

{

void TakePhoto();

}

interface IVideo

{

void MakeVideo();

}

interface IWeb

{

void BrowseInternet();

}

class Camera : IPhoto

{

public void TakePhoto()

{

Console.WriteLine("Фотографируем с помощью фотокамеры");

}

}

class Phone : ICall, IPhoto, IVideo, IWeb

{

public void Call()

{

Console.WriteLine("Звоним");

}

public void TakePhoto()

{

Console.WriteLine("Фотографируем с помощью смартфона");

}

public void MakeVideo()

{

Console.WriteLine("Снимаем видео");

}

public void BrowseInternet()

{

Console.WriteLine("Серфим в интернете");

}

}

Для применения принципа разделения интерфейсов, интерфейс класса Phone разделяется на группы связанных методов (в данном случае получается 4 группы, в каждой по одному методу). Затем каждая группа обертывается в отдельный интерфейс и используется самостоятельно.

**D - Принцип инверсии зависимостей** (Dependency Inversion Principle) служит для создания слабосвязанных сущностей, которые легко тестировать, модифицировать и обновлять. Этот принцип можно сформулировать следующим образом:

**Модули верхнего уровня не должны зависеть от модулей нижнего уровня. И те и другие должны зависеть от абстракций.**

**Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.**

Чтобы понять принцип, рассмотрим следующий пример:

class Book

{

public string Text { get; set; }

public ConsolePrinter Printer { get; set; }

public void Print()

{

Printer.Print(Text);

}

}

class ConsolePrinter

{

public void Print(string text)

{

Console.WriteLine(text);

}

}

Класс Book, представляющий книгу, использует для печати класс ConsolePrinter. При подобном определении класс Book зависит от класса ConsolePrinter. Более того мы жестко определили, что печать книгу можно только на консоли с помощью класса ConsolePrinter. Другие же варианты, например, вывод на принтер, вывод в файл или с использованием каких-то элементов графического интерфейса - все это в данном случае исключено. Абстракция печати книги не отделена от деталей класса ConsolePrinter. Все это является нарушением принципа инверсии зависимостей.

Теперь попробуем привести наши классы в соответствие с принципом инверсии зависимостей, отделив абстракции от низкоуровневой реализации:

interface IPrinter

{

void Print(string text);

}

class Book

{

public string Text { get; set; }

public IPrinter Printer { get; set; }

public Book(IPrinter printer)

{

this.Printer = printer;

}

public void Print()

{

Printer.Print(Text);

}

}

class ConsolePrinter : IPrinter

{

public void Print(string text)

{

Console.WriteLine("Печать на консоли");

}

}

class HtmlPrinter : IPrinter

{

public void Print(string text)

{

Console.WriteLine("Печать в html");

}

}

Теперь абстракция печати книги отделена от конкретных реализаций. В итоге и класс Book и класс ConsolePrinter зависят от абстракции IPrinter. Кроме того, теперь мы также можем создать дополнительные низкоуровневые реализации абстракции IPrinter и динамически применять их в программе:

Book book = new Book(new ConsolePrinter());

book.Print();

book.Printer = new HtmlPrinter();

book.Print();

SOA — это набор архитектурных принципов, не зависящих от технологий и продуктов, совсем как полиморфизм или инкапсуляция.

**SOA (Service Oriented Architecture)** - концепция сервис-ориентированной архитектуры, предназначенная для решения вопросов интеграции информационной инфраструктуры компании за счет построения архитектуры, позволяющей интегрировать с максимальной гибкостью разнородные приложения.

Сервис-ориентированная архитектура строится за счет проектирования и разработки сервисов и средств их подключения. Сервис представляет собой определенную работу или бизнес-функцию, предназначенную для обеспечения согласованной работы приложений.

SOA не зависит от языков программирования, платформ или протокольных спецификаций, с помощью которых сервисы разрабатываются. В частности принципами SOA являются:

* Архитектура не привязана к определённой технологии
* Независимость организации системы от используемых платформ
* Независимость организации системы от применяемых языков программирования
* Использование сервисов, независимых от конкретных приложений, с единообразными интерфейсами доступа к ним
* Организация сервисов как слабосвязанных компонент для построения систем

Появление SOA обусловлено желанием перейти от программирования комплексных информационных продуктов к возможности «сборки» удовлетворяющей информационные потребности системы из разнородных приложений. Так же не маловажным аспектом SOA является высокая гибкость, которая достигается за счет возможности быстрой корректировки бизнес-логики - изменение, вносимое в бизнес-функцию, в итоге затронет все необходимые приложения.

С точки зрения построения архитектуры, SOA включает следующие ключевые элементы:

* приложение-клиент, как правило, отвечающее за инициацию бизнес-процесса и получающее результаты его выполнения
* сервис – программный компонент, имеющий четко определенную функциональность и соответствующий бизнес-задаче
* репозиторий сервисов, предоставляющий механизмы для обнаружения сервисов и получения дополнительной информации о них
* сервисная шина, обеспечивающая взаимодействие между всеми компонентами архитектуры

Так как сама по себе SOA не является продуктом или технологией, но стилем архитектуры, развертывание SOA подразумевает использование различных существующих продуктов и технологий. Например, такие языки как [BPEL](https://piter-soft.ru/knowledge/glossary/process/bpel-system.html), расширяют концепцию SOA, предоставляя метод объединения мелких сервисов в более обширные бизнес-сервисы, включаемые в состав автоматизированных бизнес-процессов. Таким образом, BPMS определяет правила, согласно которым будут вызываться сервисы и передаваться информацию между ними, что объясняет возросшую популярность совместного использования SOA и [BPM](https://piter-soft.ru/knowledge/glossary/process/bpm-system.html).

Паттерны проектирования:

**Порождающие паттерны**

Порождающие паттерны — это паттерны, которые абстрагируют процесс инстанцирования или, иными словами, процесс порождения классов и объектов.

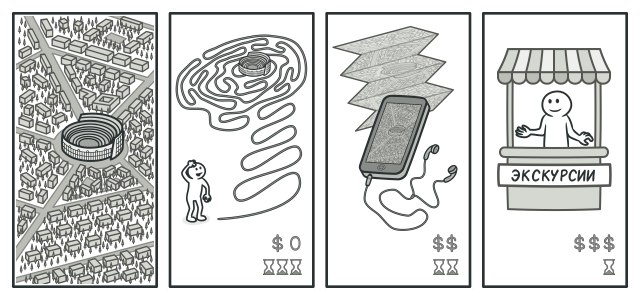
Другая группа паттернов - **структурные паттерны** - рассматривает, как классы и объекты образуют более крупные структуры - более сложные по характеру классы и объекты.

Третья группа паттернов называются **поведенческими** - они определяют алгоритмы и взаимодействие между классами и объектами, то есть их поведение.

## 17. Приемы объектно-ориентированного программирования. Итератор.

Паттерн Итератор (Iterator) предоставляет абстрактный интерфейс для последовательного доступа ко всем элементам составного объекта без раскрытия его внутренней структуры.

Ключевым моментом, который позволяет осуществить перебор коллекций с помощью foreach, является применения этими классами коллекций паттерна итератор, или проще говоря пары интерфейсов IEnumerable / IEnumerator. Интерфейс IEnumerator определяет функционал для перебора внутренних объектов в контейнере.



**Когда использовать итераторы?**

* Когда необходимо осуществить обход объекта без раскрытия его внутренней структуры
* Когда имеется набор составных объектов, и надо обеспечить единый интерфейс для их перебора
* Когда необходимо предоставить несколько альтернативных вариантов перебора одного и того же объекта

class Client

{

    public void Main()

    {

        Aggregate a = new ConcreteAggregate();

        Iterator i = a.CreateIterator();

        object item = i.First();

        while (!i.IsDone())

        {

            item = i.Next();

        }

    }

}

abstract class Aggregate

{

    public abstract Iterator CreateIterator();

    public abstract int Count { get; protected set; }

    public abstract object this[int index] { get; set; }

}

class ConcreteAggregate : Aggregate

{

    private readonly ArrayList \_items = new ArrayList();

    public override Iterator CreateIterator()

    {

        return new ConcreteIterator(this);

    }

    public override int Count

    {

        get { return \_items.Count; }

        protected set { }

    }

    public override object this[int index]

    {

        get { return \_items[index]; }

        set { \_items.Insert(index, value); }

    }

}

abstract class Iterator

{

    public abstract object First();

    public abstract object Next();

    public abstract bool IsDone();

    public abstract object CurrentItem();

}

class ConcreteIterator : Iterator

{

    private readonly Aggregate \_aggregate;

    private int \_current;

    public ConcreteIterator(Aggregate aggregate)

    {

        this.\_aggregate = aggregate;

    }

    public override object First()

    {

        return \_aggregate[0];

    }

    public override object Next()

    {

        object ret = null;

        \_current++;

        if (\_current < \_aggregate.Count)

        {

            ret = \_aggregate[\_current];

        }

        return ret;

    }

    public override object CurrentItem()

    {

        return \_aggregate[\_current];

    }

    public override bool IsDone()

    {

        return \_current >= \_aggregate.Count;

    }

}

**Участники**

* **Iterator**: определяет интерфейс для обхода составных объектов
* **Aggregate**: определяет интерфейс для создания объекта-итератора
* **ConcreteIterator**: конкретная реализация итератора для обхода объекта Aggregate. Для фиксации индекса текущего перебираемого элемента использует целочисленную переменную \_current
* **ConcreteAggregate**: конкретная реализация Aggregate. Хранит элементы, которые надо будет перебирать
* **Client**: использует объект Aggregate и итератор для его обхода

**Преимущества**

* Упрощает классы хранения данных.
* Позволяет реализовать различные способы обхода структуры данных.
* Позволяет одновременно перемещаться по структуре данных в разные стороны.

**и недостатки**

* Не оправдан, если можно обойтись простым циклом.

## 18. Приемы объектно-ориентированного программирования. Одиночка.

Одиночка (Singleton, Синглтон) - порождающий паттерн, который гарантирует, что для определенного класса будет создан только один объект, а также предоставит к этому объекту точку доступа.

**Когда надо использовать Синглтон?** Когда необходимо, чтобы для класса существовал только один экземпляр

Синглтон позволяет создать объект только при его необходимости. Если объект не нужен, то он не будет создан. В этом отличие синглтона от глобальных переменных.

class Singleton

{

    private static Singleton instance;

    private Singleton()

    {}

    public static Singleton getInstance()

    {

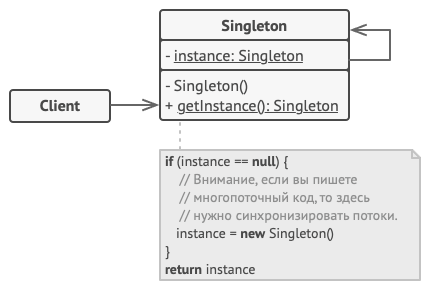
        if (instance == null)

            instance = new Singleton();

        return instance;

    }

}



В классе определяется статическая переменная - ссылка на конкретный экземпляр данного объекта и приватный конструктор. В статическом методе getInstance() этот конструктор вызывается для создания объекта, если, конечно, объект отсутствует и равен null.

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.

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.

**Преимущества**

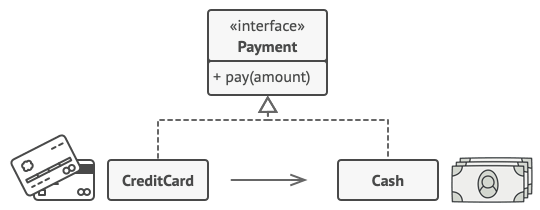
* Гарантирует наличие единственного экземпляра класса.
* Предоставляет к нему глобальную точку доступа.
* Реализует отложенную инициализацию объекта-одиночки.

**Недостатки**

* Нарушает *принцип единственной ответственности класса*.
* Маскирует плохой дизайн.
* Проблемы мультипоточности.
* Требует постоянного создания Mock-объектов при юнит-тестировании.

## 19. Приемы объектно-ориентированного программирования. Заместитель.

Паттерн Заместитель (Proxy) предоставляет объект-заместитель, который управляет доступом к другому объекту. То есть создается объект-суррогат, который может выступать в роли другого объекта и замещать его.



**Когда использовать прокси?**

* Когда надо осуществлять взаимодействие по сети, а объект-проси должен имитировать поведения объекта в другом адресном пространстве. Использование прокси позволяет снизить накладные издержки при передачи данных через сеть. Подобная ситуация еще называется **удалённый заместитель (remote proxies)**
* Когда нужно управлять доступом к ресурсу, создание которого требует больших затрат. Реальный объект создается только тогда, когда он действительно может понадобится, а до этого все запросы к нему обрабатывает прокси-объект. Подобная ситуация еще называется **виртуальный заместитель (virtual proxies)**
* Когда необходимо разграничить доступ к вызываемому объекту в зависимости от прав вызывающего объекта. Подобная ситуация еще называется **защищающий заместитель (protection proxies)**
* Когда нужно вести подсчет ссылок на объект или обеспечить потокобезопасную работу с реальным объектом. Подобная ситуация называется **"умные ссылки" (smart reference)**

class Client

{

void Main()

{

Subject subject = new Proxy();

subject.Request();

}

}

abstract class Subject

{

public abstract void Request();

}

class RealSubject : Subject

{

public override void Request()

{}

}

class Proxy : Subject

{

RealSubject realSubject;

public override void Request()

{

if (realSubject == null)

realSubject = new RealSubject();

realSubject.Request();

}

}

**Участники паттерна**

* **Subject**: определяет общий интерфейс для Proxy и RealSubject. Поэтому Proxy может использоваться вместо RealSubject
* **RealSubject**: представляет реальный объект, для которого создается прокси
* **Proxy**: заместитель реального объекта. Хранит ссылку на реальный объект, контролирует к нему доступ, может управлять его созданием и удалением. При необходимости Proxy переадресует запросы объекту RealSubject
* **Client**: использует объект Proxy для доступа к объекту RealSubject

**Преимущества**

* Позволяет контролировать сервисный объект незаметно для клиента.
* Может работать, даже если сервисный объект ещё не создан.
* Может контролировать жизненный цикл служебного объекта.

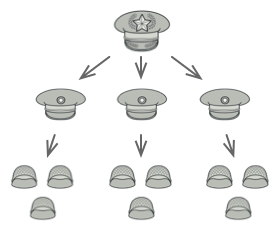
**Недостатки**

* Усложняет код программы из-за введения дополнительных классов.
* Увеличивает время отклика от сервиса.

## 20. Приемы объектно-ориентированного программирования. Компоновщик.

Паттерн Компоновщик (Composite) объединяет группы объектов в древовидную структуру по принципу "часть-целое и позволяет клиенту одинаково работать как с отдельными объектами, так и с группой объектов.

Образно реализацию паттерна можно представить в виде меню, которое имеет различные пункты. Эти пункты могут содержать подменю, в которых, в свою очередь, также имеются пункты. То есть пункт меню служит с одной стороны частью меню, а с другой стороны еще одним меню. В итоге мы однообразно можем работать как с пунктом меню, так и со всем меню в целом.



**Когда использовать компоновщик?**

* Когда объекты должны быть реализованы в виде иерархической древовидной структуры
* Когда клиенты единообразно должны управлять как целыми объектами, так и их составными частями. То есть целое и его части должны реализовать один и тот же интерфейс

class Client

{

    public void Main()

    {

        Component root = new Composite("Root");

        Component leaf = new Leaf("Leaf");

        Composite subtree = new Composite("Subtree");

        root.Add(leaf);

        root.Add(subtree);

        root.Display();

    }

}

abstract class Component

{

    protected string name;

    public Component(string name)

    {

        this.name = name;

    }

    public abstract void Display();

    public abstract void Add(Component c);

    public abstract void Remove(Component c);

}

class Composite : Component

{

    List<Component> children = new List<Component>();

    public Composite(string name)

        : base(name)

    {}

    public override void Add(Component component)

    {

        children.Add(component);

    }

    public override void Remove(Component component)

    {

        children.Remove(component);

    }

    public override void Display()

    {

        Console.WriteLine(name);

        foreach (Component component in children)

        {

            component.Display();

        }

    }

}

class Leaf : Component

{

    public Leaf(string name)

        : base(name)

    {}

    public override void Display()

    {

        Console.WriteLine(name);

    }

    public override void Add(Component component)

    {

        throw new NotImplementedException();

    }

    public override void Remove(Component component)

    {

        throw new NotImplementedException();

    }

}

**Участники**

* **Component**: определяет интерфейс для всех компонентов в древовидной структуре
* **Composite**: представляет компонент, который может содержать другие компоненты и реализует механизм для их добавления и удаления
* **Leaf**: представляет отдельный компонент, который не может содержать другие компоненты
* **Client**: клиент, который использует компоненты

**Преимущества**

* Упрощает архитектуру клиента при работе со сложным деревом компонентов.
* Облегчает добавление новых видов компонентов.

**Недостатки**

* Создаёт слишком общий дизайн классов.

## 21. Приемы объектно-ориентированного программирования. Мост.

**Определение из гугла:**

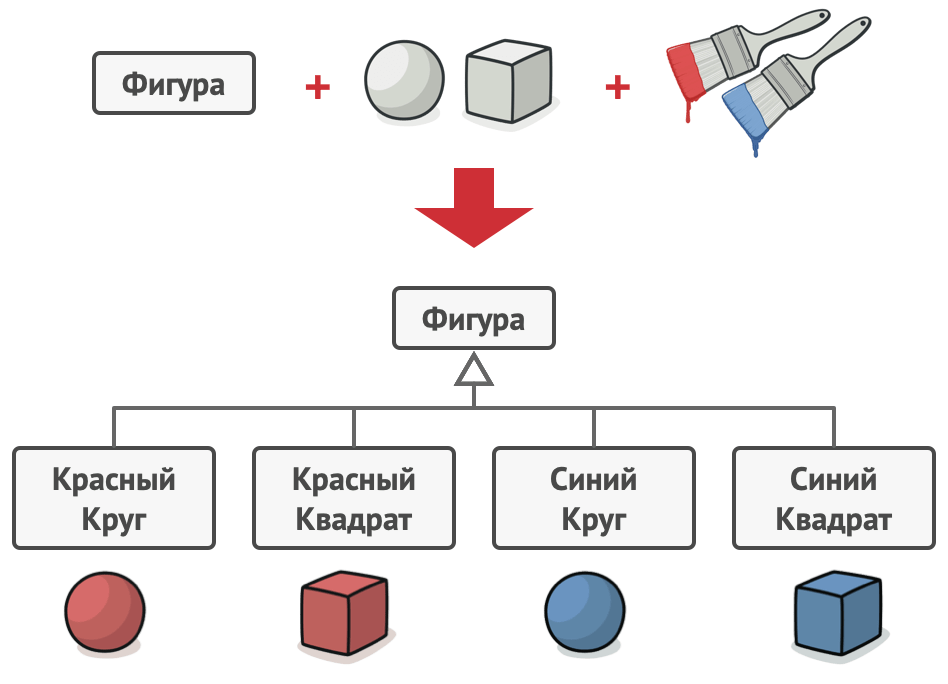
**Мост** — это структурный паттерн проектирования, который разделяет один или несколько классов на две отдельные иерархии — абстракцию и реализацию, позволяя изменять их независимо друг от друга.

**Определение из презентаций:**

**Мост** – делегирование функциональности метода другому объекту через интерфейс, чтобы иметь возможность независимо менять реализацию интерфейса.

**Проблема.**

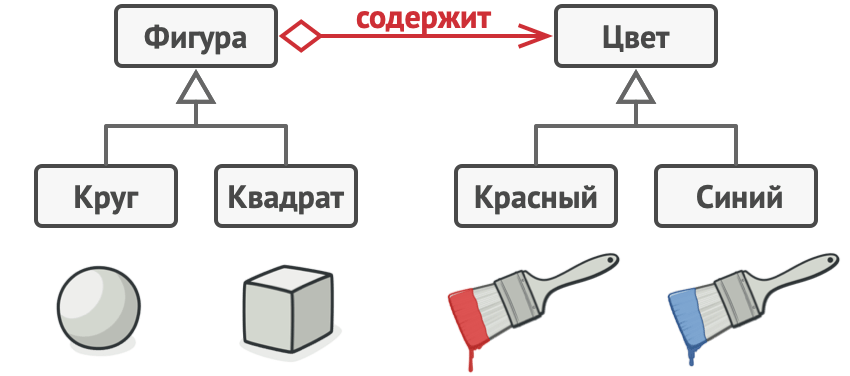
У вас есть класс геометрических *Фигур*, который имеет подклассы *Круг* и *Квадрат*. Вы хотите расширить иерархию фигур по цвету, то есть иметь *Красные* и *Синие* фигуры. Но чтобы всё это объединить, вам придётся создать 4 комбинации подклассов, вроде *СиниеКруги* и *КрасныеКвадраты*.



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.**

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.



**Реализация.**

Реализацию рассмотрим на примере устройств, управляющихся с пульта. Где есть абстрактный класс девайса и есть абстрактный класс пульта, которые можно расширять в зависимости от функционала девайсов (для радио, например, актуальны будут кнопки поиска частоты вещания, а для кондиционера – настройка температуры и типа обдува).

Ниже представлен псевдокод, но все довольно понятно.

// Класс пультов имеет ссылку на устройство, которым управляет.

// Методы этого класса делегируют работу методам связанного

// устройства.

class Remote is

protected field device: Device

constructor Remote(device: Device) is

this.device = device

method togglePower() is

if (device.isEnabled()) then

device.disable()

else

device.enable()

method volumeDown() is

device.setVolume(device.getVolume() - 10)

method volumeUp() is

device.setVolume(device.getVolume() + 10)

method channelDown() is

device.setChannel(device.getChannel() - 1)

method channelUp() is

device.setChannel(device.getChannel() + 1)

// Вы можете расширять класс пультов, не трогая код устройств.

class AdvancedRemote extends Remote is

method mute() is

device.setVolume(0)

// Все устройства имеют общий интерфейс. Поэтому с ними может

// работать любой пульт.

interface Device is

method isEnabled()

method enable()

method disable()

method getVolume()

method setVolume(percent)

method getChannel()

method setChannel(channel)

// Но каждое устройство имеет особую реализацию.

class Tv implements Device is

// ...

class Radio implements Device is

// ...

// Где-то в клиентском коде.

tv = new Tv()

remote = new Remote(tv)

remote.togglePower()

radio = new Radio()

remote = new AdvancedRemote(radio)

Если код вышел был непонятен (F тогда), то ниже ссылка на реализацию на Вашем любимом языке:

Код выше (рекомендуется):  
[Java](https://refactoring.guru/ru/design-patterns/bridge/java/example)  
Пример реализации паттерна:  
[C#](https://refactoring.guru/ru/design-patterns/bridge/csharp/example), [Python](https://refactoring.guru/ru/design-patterns/bridge/python/example)

## 22. Приемы объектно-ориентированного программирования. Наблюдатель.

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

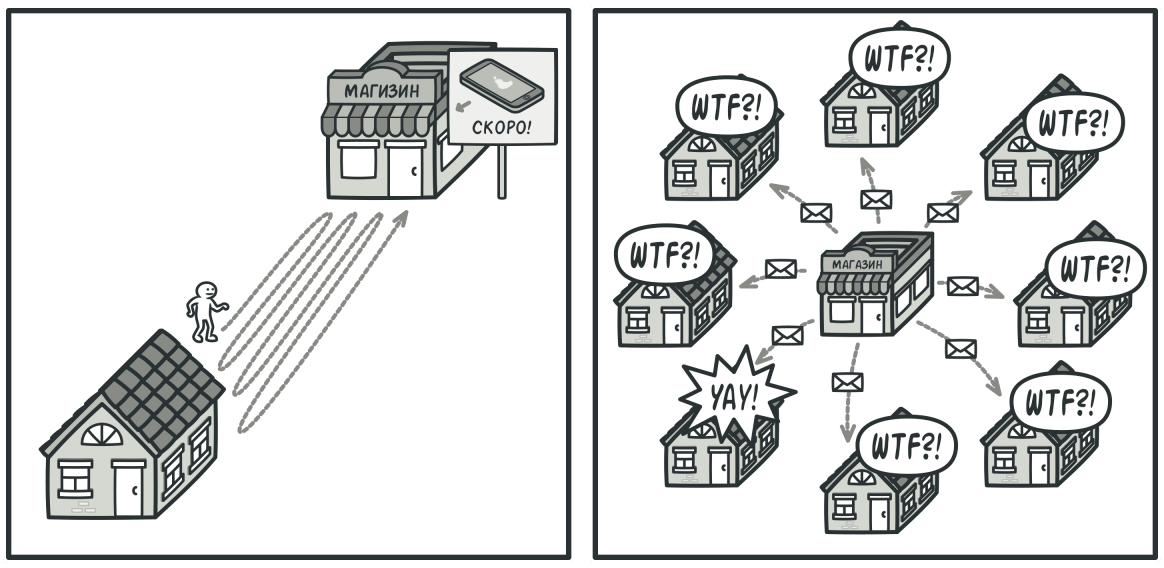
**Проблема.**

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.

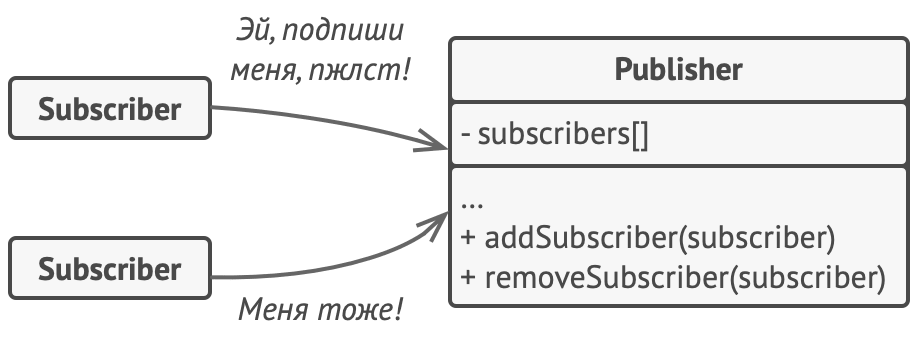
Получается конфликт: либо покупатель тратит время на периодические проверки, либо магазин тратит ресурсы на бесполезные оповещения.



**Решение.**

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

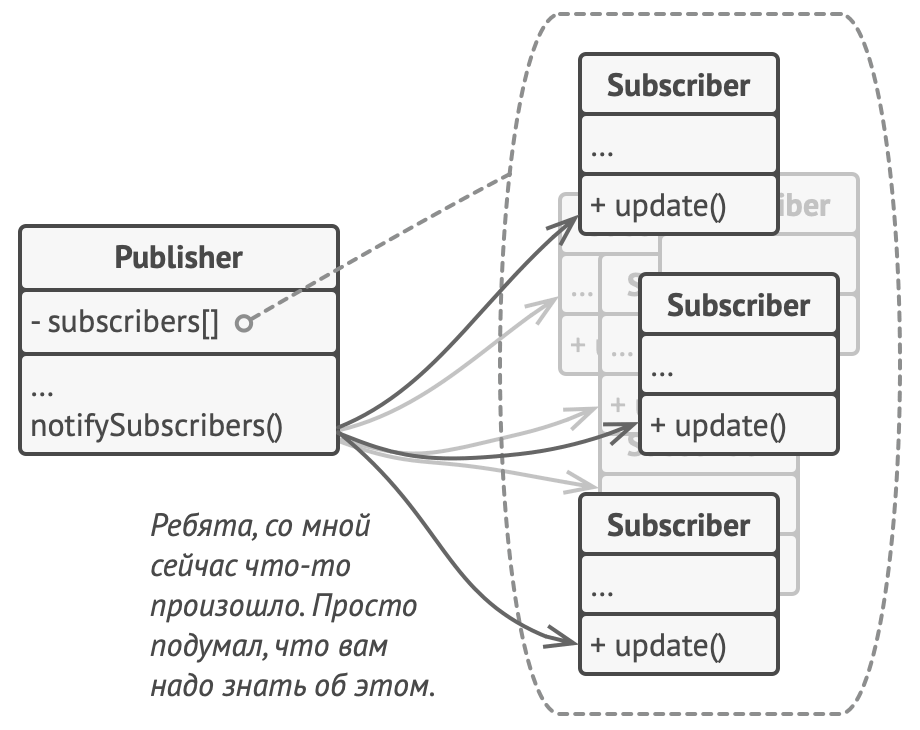
Паттерн Наблюдатель предлагает хранить внутри объекта издателя список ссылок на объекты подписчиков, причём издатель не должен вести список подписки самостоятельно. Он предоставит методы, с помощью которых подписчики могли бы добавлять или убирать себя из списка.



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.



**Реализация.**

Ниже представлен псевдокод, но все довольно понятно.

// Базовый класс-издатель. Содержит код управления подписчиками

// и их оповещения.

class EventManager is

private field listeners: hash map of event types and listeners

method subscribe(eventType, listener) is

listeners.add(eventType, listener)

method unsubscribe(eventType, listener) is

listeners.remove(eventType, listener)

method notify(eventType, data) is

foreach (listener in listeners.of(eventType)) do

listener.update(data)

// Конкретный класс-издатель, содержащий интересную для других

// компонентов бизнес-логику. Мы могли бы сделать его прямым

// потомком EventManager, но в реальной жизни это не всегда

// возможно (например, если у класса уже есть родитель). Поэтому

// здесь мы подключаем механизм подписки при помощи композиции.

class Editor is

public field events: EventManager

private field file: File

constructor Editor() is

events = new EventManager()

// Методы бизнес-логики, которые оповещают подписчиков об

// изменениях.

method openFile(path) is

this.file = new File(path)

events.notify("open", file.name)

method saveFile() is

file.write()

events.notify("save", file.name)

// ...

// Общий интерфейс подписчиков. Во многих языках, поддерживающих

// функциональные типы, можно обойтись без этого интерфейса и

// конкретных классов, заменив объекты подписчиков функциями.

interface EventListener is

method update(filename)

// Набор конкретных подписчиков. Они реализуют добавочную

// функциональность, реагируя на извещения от издателя.

class LoggingListener implements EventListener is

private field log: File

private field message

constructor LoggingListener(log\_filename, message) is

this.log = new File(log\_filename)

this.message = message

method update(filename) is

log.write(replace('%s',filename,message))

class EmailAlertsListener implements EventListener is

private field email: string

constructor EmailAlertsListener(email, message) is

this.email = email

this.message = message

method update(filename) is

system.email(email, replace('%s',filename,message))

// Приложение может сконфигурировать издателей и подписчиков как

// угодно, в зависимости от целей и окружения.

class Application is

method config() is

editor = new Editor()

logger = new LoggingListener(

"/path/to/log.txt",

"Someone has opened file: %s");

editor.events.subscribe("open", logger)

emailAlerts = new EmailAlertsListener(

"admin@example.com",

"Someone has changed the file: %s")

editor.events.subscribe("save", emailAlerts)

Если код вышел был непонятен (F тогда), то ниже ссылка на реализацию на Вашем любимом языке:

Код выше (рекомендуется):  
[Java](https://refactoring.guru/ru/design-patterns/observer/java/example)  
Пример реализации паттерна:  
[C#](https://refactoring.guru/ru/design-patterns/observer/csharp/example), [Python](https://refactoring.guru/ru/design-patterns/observer/python/example)

## 23. Приемы объектно-ориентированного программирования. Посетитель.

**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, например:

class ExportVisitor implements Visitor is

method doForCity(City c) { ... }

method doForIndustry(Industry f) { ... }

method doForSightSeeing(SightSeeing ss) { ... }

// ...

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.

foreach (Node node in graph)

if (node instanceof City)

exportVisitor.doForCity((City) node)

if (node instanceof Industry)

exportVisitor.doForIndustry((Industry) node)

// ...

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**. Вместо того, чтобы самим искать нужный метод, мы можем поручить это объектам, которые передаём в параметрах посетителю. А они уже вызовут правильный метод посетителя.

// Client code

foreach (Node node in graph)

node.accept(exportVisitor)

// City

class City is

method accept(Visitor v) is

v.doForCity(this)

// ...

// Industry

class Industry is

method accept(Visitor v) is

v.doForIndustry(this)

// ...

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.

**Реализация.**

Ниже представлен псевдокод, но все довольно понятно.

// Сложная иерархия элементов.

interface Shape is

method move(x, y)

method draw()

method accept(v: Visitor)

// Метод принятия посетителя должен быть реализован в каждом

// элементе, а не только в базовом классе. Это поможет программе

// определить, какой метод посетителя нужно вызвать, если вы не

// знаете тип элемента.

class Dot implements Shape is

// ...

method accept(v: Visitor) is

v.visitDot(this)

class Circle implements Shape is

// ...

method accept(v: Visitor) is

v.visitCircle(this)

class Rectangle implements Shape is

// ...

method accept(v: Visitor) is

v.visitRectangle(this)

class CompoundShape implements Shape is

// ...

method accept(v: Visitor) is

v.visitCompoundShape(this)

// Интерфейс посетителей должен содержать методы посещения

// каждого элемента. Важно, чтобы иерархия элементов менялась

// редко, так как при добавлении нового элемента придётся менять

// всех существующих посетителей.

interface Visitor is

method visitDot(d: Dot)

method visitCircle(c: Circle)

method visitRectangle(r: Rectangle)

method visitCompoundShape(cs: CompoundShape)

// Конкретный посетитель реализует одну операцию для всей

// иерархии элементов. Новая операция = новый посетитель.

// Посетитель выгодно применять, когда новые элементы

// добавляются очень редко, а новые операции — часто.

class XMLExportVisitor implements Visitor is

method visitDot(d: Dot) is

// Экспорт id и координат центра точки.

method visitCircle(c: Circle) is

// Экспорт id, кординат центра и радиуса окружности.

method visitRectangle(r: Rectangle) is

// Экспорт id, кординат левого-верхнего угла, ширины

// и высоты прямоугольника.

method visitCompoundShape(cs: CompoundShape) is

// Экспорт id составной фигуры, а также списка id

// подфигур, из которых она состоит.

// Приложение может применять посетителя к любому набору

// объектов элементов, даже не уточняя их типы. Нужный метод

// посетителя будет выбран благодаря проходу через метод accept.

class Application is

field allShapes: array of Shapes

method export() is

exportVisitor = new XMLExportVisitor()

foreach (shape in allShapes) do

shape.accept(exportVisitor)

Если код вышел был непонятен (F тогда), то ниже ссылка на реализацию на Вашем любимом языке:

Код выше (рекомендуется):  
[Java](https://refactoring.guru/ru/design-patterns/visitor/java/example)  
Пример реализации паттерна:  
[C#](https://refactoring.guru/ru/design-patterns/visitor/csharp/example), [Python](https://refactoring.guru/ru/design-patterns/visitor/python/example)

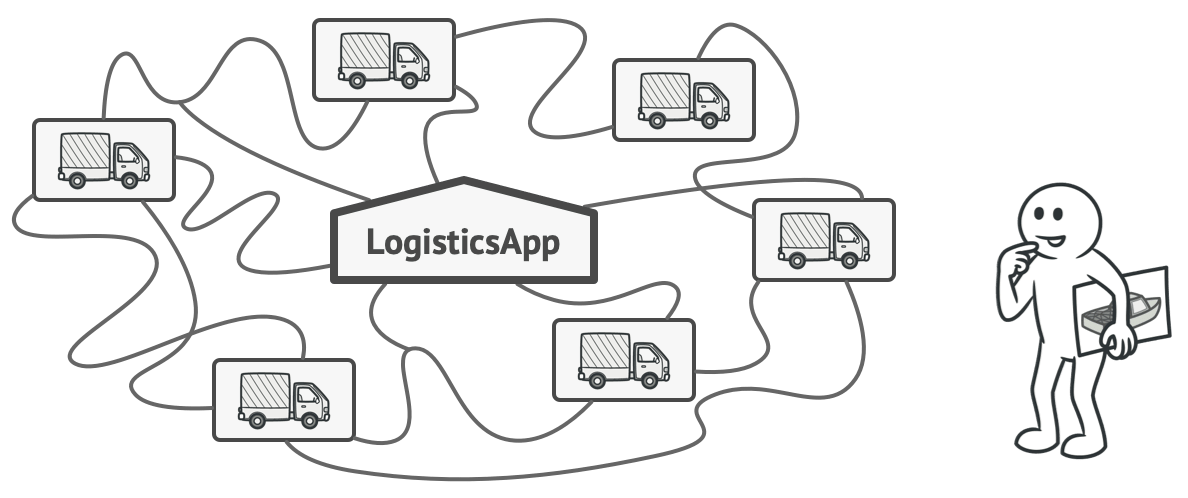
## 24. Приемы объектно-ориентированного программирования. Фабричный метод. Фабрика классов.

**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.

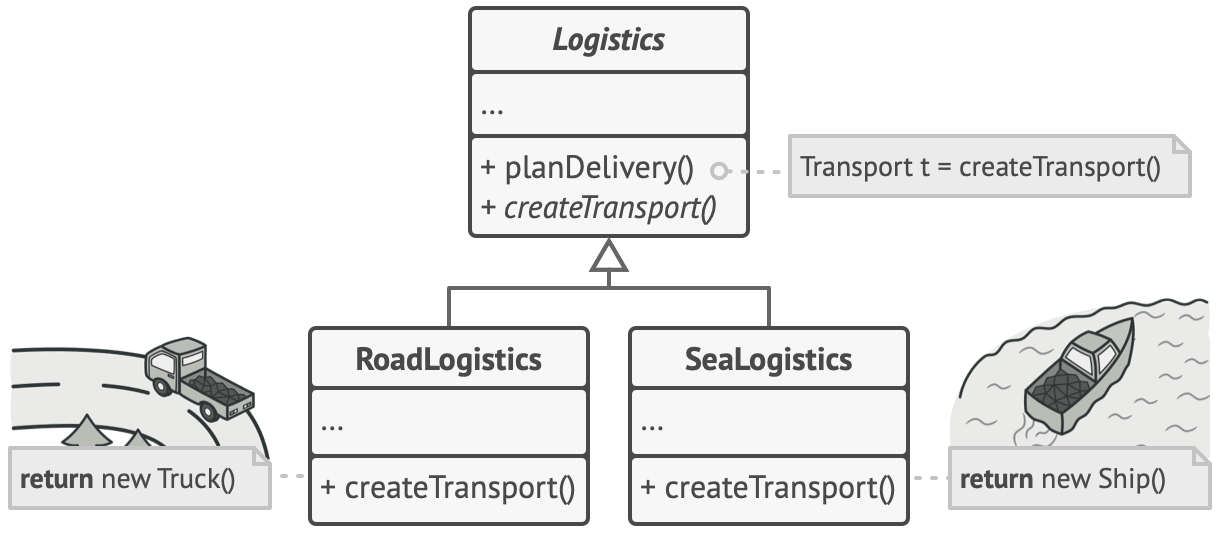


Отличные новости, правда?! Но как насчёт кода? Большая часть существующего кода жёстко привязана к классам *Грузовиков*. Чтобы добавить в программу классы морских *Судов*, понадобится перелопатить всю программу. Более того, если вы потом решите добавить в программу ещё один вид транспорта, то всю эту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

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

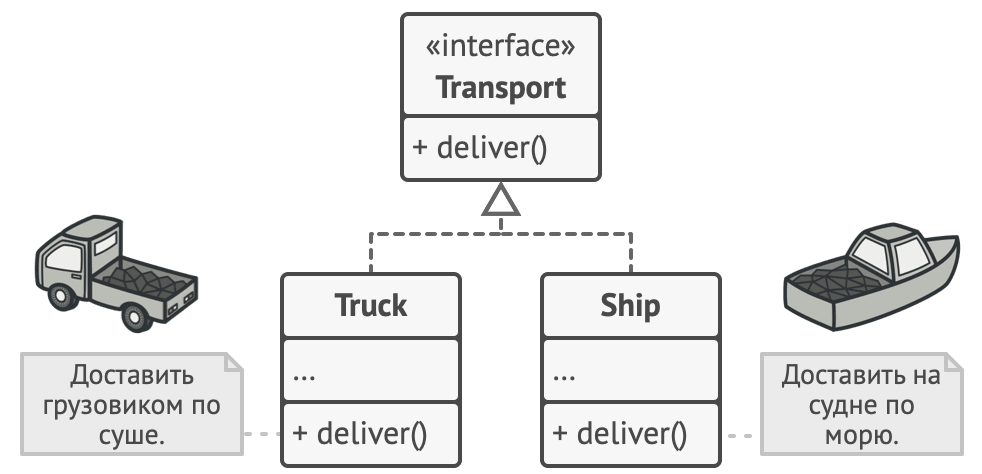
**Решение.**

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.

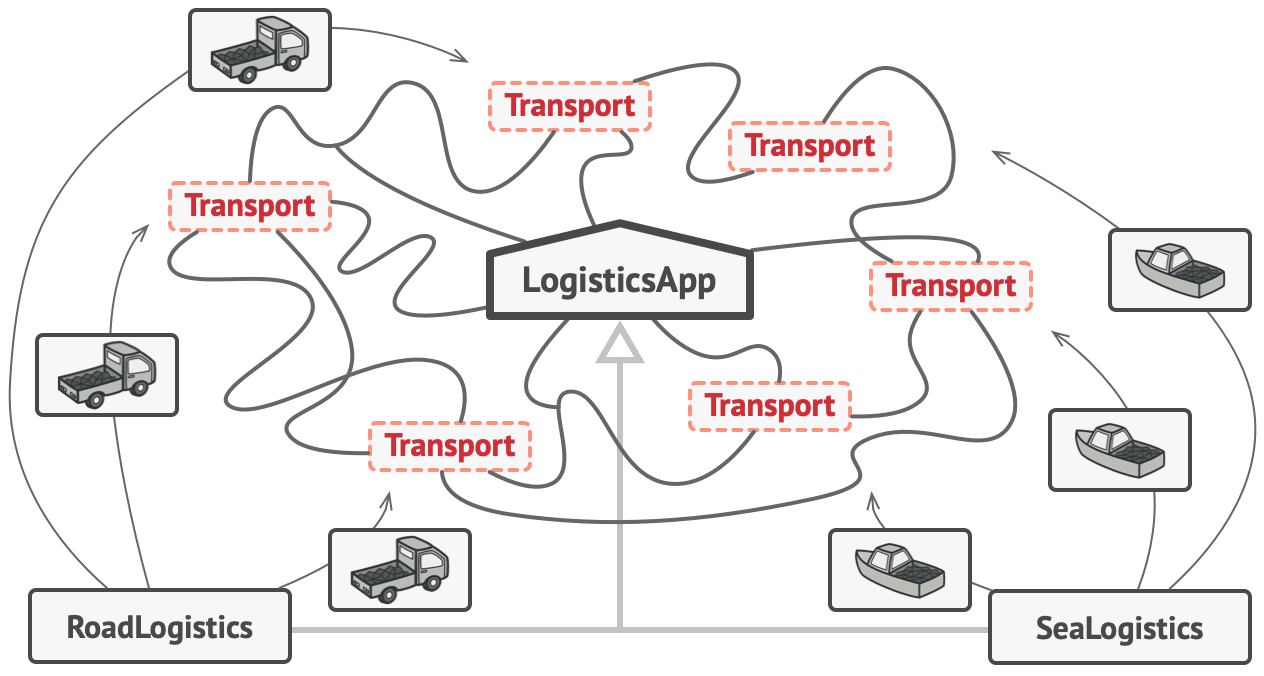


На первый взгляд, это может показаться бессмысленным: мы просто переместили вызов конструктора из одного конца программы в другой. Но теперь вы сможете переопределить фабричный метод в подклассе, чтобы изменить тип создаваемого продукта.

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.

**Реализация.**

Ниже представлен псевдокод, но все довольно понятно.

// Паттерн Фабричный метод применим тогда, когда в программе

// есть иерархия классов продуктов.

interface Button is

method render()

method onClick(f)

class WindowsButton implements Button is

method render(a, b) is

// Отрисовать кнопку в стиле Windows.

method onClick(f) is

// Навесить на кнопку обработчик событий Windows.

class HTMLButton implements Button is

method render(a, b) is

// Вернуть HTML-код кнопки.

method onClick(f) is

// Навесить на кнопку обработчик события браузера.

// Базовый класс фабрики. Заметьте, что "фабрика" — это всего

// лишь дополнительная роль для класса. Скорее всего, он уже

// имеет какую-то бизнес-логику, в которой требуется создание

// разнообразных продуктов.

class Dialog is

method render() is

// Чтобы использовать фабричный метод, вы должны

// убедиться в том, что эта бизнес-логика не зависит от

// конкретных классов продуктов. Button — это общий

// интерфейс кнопок, поэтому все хорошо.

Button okButton = createButton()

okButton.onClick(closeDialog)

okButton.render()

// Мы выносим весь код создания продуктов в особый метод,

// который назвают "фабричным".

abstract method createButton():Button

// Конкретные фабрики переопределяют фабричный метод и

// возвращают из него собственные продукты.

class WindowsDialog extends Dialog is

method createButton():Button is

return new WindowsButton()

class WebDialog extends Dialog is

method createButton():Button is

return new HTMLButton()

class Application is

field dialog: Dialog

// Приложение создаёт определённую фабрику в зависимости

// от конфигурации или окружения.

method initialize() is

config = readApplicationConfigFile()

if (config.OS == "Windows") then

dialog = new WindowsDialog()

else if (config.OS == "Web") then

dialog = new WebDialog()

else

throw new Exception("Error! Unknown operating system.")

// Если весь остальной клиентский код работает с фабриками

// и продуктами только через общий интерфейс, то для него

// будет не важно, какая фабрика была создана изначально.

method main() is

this.initialize()

dialog.render()

Если код вышел был непонятен (F тогда), то ниже ссылка на реализацию на Вашем любимом языке:

Код выше (рекомендуется):  
[Java](https://refactoring.guru/ru/design-patterns/factory-method/java/example)  
Пример реализации паттерна:  
[C#](https://refactoring.guru/ru/design-patterns/factory-method/csharp/example), [Python](https://refactoring.guru/ru/design-patterns/factory-method/python/example)

## 26. Имитация модульного программирования в языке C++. Пространство имен.

1.1. Принципы модульного программирования на языке С++

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 этого модуля. 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.

Очевидно, что программисту при включении h-файла другого модуля предоставляется выбор: подключить его в h-файле модуля или в cpp-файле. В данном случае предпочтение следует отдавать части реализации модуля (cpp-файл).

При подключении h-файла следует придерживаться следующей схемы: предположим, что наш модуль называется SysModuleи состоит из двух частей: SysModule.h и SysModule.cpp. Рекомендуется следующая схема подключения:

SysModule.h:

#include "Config.h" // наш файл конфигурации

// подключается первым во всех h-файлах

// всех наших проектов

#include "Другой стандартный модуль"

#include "Другой наш модуль"

SysModule.cpp:

#include "Файл предкомпилированных заголовков"

#include "Еще один наш модуль"

#include "Другой стандартный модуль"

#include "SysModule.h" // подключается последним

Поскольку один и тот же h-файл может одновременно включатся в другие h-файлы и несколько раз подключаться при компиляции одного и того же cpp-файла, его следует защищать от повторной компиляции. Для этого в начале любого h-файла вставляются следующие директивы компилятора:

#ifndef \_\_SysModule\_h\_\_

#define \_\_SysModule\_h\_\_

...

#endif //\_\_SysModule\_h\_\_

Таким образом, в том случае, когда файл подключается несколько раз, скомпилируется он только один раз.

Внимание! Согласно стандарту ISO, любой h- и cpp-файл в С++ должен заканчиваться символом перевода строки.

1.2. Пространства имен

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

namespace Sys

{

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 внутри модуля следует помещать в пространство имен. Следует давать небольшой буквенный идентификатор, который будет соответствовать префиксу файла.

Существует возможность открыть пространство имен таким образом, чтобы можно было использовать неполную форму записи. Для этого надо написать строку:

using namespace Sys;

Но следует отметить, что данная конструкция является причиной многих ошибок, поэтому так писать не стоит.

Существует второй способ открыть пространство имен — это открыть его для конкретного определения:

using Sys::Proc();

...

Proc();

...

Но рекомендуется использовать Sys::Proc();

Идентификаторы, объявленные вне пространства имен, относятся к так называемому глобальному пространству имен, доступ к которым осуществляется с помощью оператора **::**

::Funk();

Для того чтобы была возможность закрыть доступ к данным и подпрограммам внутри данного пространства существует пространство имен без имени:

namespace

{

...

}

Пространства имен могут быть вложенными:

namespace Sys

{

namespace Local

{

int var;

...

}

...

}

Sys::Local::var = 10;

Замечание! Когда возникает желание объявить переменный тип данных или подпрограмму внутри пространства имен, а реализовать за пределами (или наоборот), следует поступать так:

SysModul.h: SysModul.cpp:

namespace Sys namespace Sys

{ {

int Proc(); int Proc();

} {

...

};

}

## 27. Классы и объекты в языке C++. Классы в языке C++. Наследование. Конструкторы и деструкторы. Стандартные конструкторы. Создание объектов по значению (на стеке) и по ссылке (в динамической памяти). Операторы new и delete. Размещающий оператор new. Порядок конструирования и разрушения объектов. Вложенные определения классов. «Друзья» класса. Статические члены класса.

* **Классы в языке С++**

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

class TTextReader struct TTextReader

{ {

... // private ... // public

}; };

В С++ доступны атрибуты доступа в классах:

* public
* protected
* private

В Delphi данные секции принято употреблять в порядке: **private...protected...public**. В С++ их можно чередовать.

В работе секций protected и private в Delphi и C++ есть различия:

* В Delphi классы внутри одного модуля могут обращаться к данным и подпрограммам друг друга без ограничений. А действие секций protected и private распространяется только за пределами данного модуля.
* В С++ действие этих секций распространяется на любые два класса. Но установленные ограничения можно обойти с помощью специального оператора **friend**:

class TTextReader

{

friend class TList;

};

После этого объект класса TList может обращаться к полям из секций private и protected класса TTextReader.

* **Наследование**

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.

* **Конструкторы и деструкторы**

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

class TTextReader

{

public:

TTextReader();

~TTextReader();

};

В Delphi стандартный деструктор является виртуальным. В С++ это определяет программист, если планируется создавать объекты в динамической памяти (по ссылке), деструктор необходимо делать виртуальным:

virtual ~TTextReader();

**Создание объектов:**

* по значению (на стеке):

TTextReader reader;

* по ссылке (в динамической памяти):

TTextReader\*reader = new TTextReader(); //оператор new служит для размещения объекта в динамической памяти

* с помощью оператора new:

TTextReader\*reader = new (адрес ) TTextReader;

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 вызове деструктора

* 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.

Так выглядит динамическое создание и разрушение объектов:

new:

malloc();

TTextReader();

delete:

~TTextReader();

free();

* **Стандартные конструкторы**

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

* конструктор без параметров
* конструктор копирования

Пример:

class TTextReader

{

public:

TTextReader(); // конструктор без параметров

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

}

Внимание! Если программист определил хотя бы один конструктор в класс — компилятор не создаст никаких стандартных конструкторов.

Конструктор без параметров создается для того, чтобы можно было написать:

TTextReader R;

Конструктор копирования нужен для следующей записи:

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

Конструктор копирования вызывается в том случае, когда создаваемый по значению объект создается путем копирования другого уже существующего объекта.

Следует отметить, что запись:

TTextReader R1 = R2;

и два оператора:

TTextReader R1;

R1 = R2;

имеют схожий синтаксис, но разную семантику: в первом случае объект создается конструктором копирования, во втором — конструктором без параметров, а затем с помощью оператора ‘=’ выполняется присваивание одного объекта другому (данный вариант требует перегрузки оператора ‘=’ для класса TTextReader)

Работа стандартного конструктора копирования, создаваемого компилятором, заключается в том, чтобы выполнить полное копирование памяти с помощью функции **memcpy**.

* **Создание объектов по значению (на стеке) и по ссылке (в динамической памяти)**

Об этом в пункте про конструкторы и деструкторы :)

* **Операторы new/delete**

И это тоже там

* **Размещающий оператор new**

Обычно оператор **new** размещает объекты в динамической памяти (**heap**).

TDelimitedReader \*R = new TDelimitedReader();

В данном случае оператор new имеет следующий вид:

void \*operator new(size\_t size);

Существует вид оператора new, который позволяет расположить объект по заданному адресу:

TDelimitedReader \*R = new (Buffer)TDelimitedReader();

В этом же случае оператор new имеет следующий вид:

void \*operator new(size\_t size, void \*p)

{

return p;

}

Внимание! Комбинировать различные способы выделения и освобождения памяти не рекомендуется.

* **Порядок конструирования и разрушения объектов**

По причине автоматичности конструкторов и деструкторов в С++ существует определенный порядок конструирования базовых и агрегированных объектов.

class TTextReader

{

public:

TTextReader();

~TTextReader();

}

class TDelimitedReader: public TTextReader

{

public:

TDelimitedReader();

~TDelimitedReader();

}

TDelimitedReader::TDelimitedReader()

{

...

}

TDelimitedReader::~TDelimitedReader()

{

...

}

В конструкторе производного класса конструктор базового класса вызывается автоматически до выполнения первого оператора в теле конструктора.

В деструкторе производного класса деструктор базового класса вызывается автоматически после последнего оператора в теле деструктора.

Если базовый класс содержит конструктор с параметрами или несколько конструкторов, то возникает неопределенность в том, какой конструктор базового класса будет вызван. Эту неопределенность можно устранить следующим образом:

TDelimitedReader::TDelimitedReader() : TTextReader(...)

// в скобках записываются параметры для вызова конструктора

{

...

}

После двоеточия разрешена запись списка операторов, разделенных запятыми. Эти операторы называются списком инициализации.

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.

* **Вложенные определения классов**

В С++ внутри класса можно определить другой класс.

class TTextReader

{

public:

class TItems

{

...

};

...

};

Эта запись по смыслу соответствует следующей записи:

class TTextReader::TItems

{

...

};

class TTextReader

{

friend class TTextReader::TItems; // см. ниже "Друзья класса"

};

Таким образом, определения классов и других типов данных внутри класса означает использование имени внешнего класса как пространства имен.

* **«Друзья» класса**

Для того, чтобы объекты некоторого класса могли получить доступ в private и protected полям другого класса, используется оператор **friend**, который разрешает доступ ко всем записям и методам класса для того класса, который указан в операторе. Данный оператор используется внутри класса, с его помощью нельзя разрешать доступ к членам класса извне, иначе это нарушит принцип сокрытия данных.

* **Статические члены класса**

Поля и методы класса могут быть объявлены при помощи слова **static**:

class TTextReader

{

public:

...

static char\*ClassName();

...

private:

static int m\_ObjectCount;

...

};

По смыслу данный код эквивалентен следующему:

class TTextReader

{

friend char\*TTextReader::ClassName();

...

};

class TTextReader::ClassName()

{

...

};

int TTextReader::m\_ObjectCount;

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. Исключения в языке C++. Обработка исключений ситуаций в языке C++. Защита от утечки ресурсов. Имитация оператора try-finally. Понятие автоматического указателя (auto\_ptr). Использование автоматических указателей для защиты от утечки ресурсов.

**Обработка исключительных ситуаций в С++**

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.

В С++ существует аналог блока **try…except…end**:

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.

**Защита ресурсов от утечки и остальное**

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.

class AutoPtr

{

public:

AutoPtr(int \*arr);

~AutoPtr();

private:

int \*m\_arr;

};

AutoPtr::AutoPtr(int \*arr)

{

m\_arr = arr;

}

AutoPtr::~AutoPtr()

{

delete[] m\_arr;

}

void Proc()

{

int \*arr = new int[100];

AutoPtr autoc(arr);

...

}

## 29. Множественное наследование в языке C++. Множественное наследование. Проблема повторяющихся базовых классов. Замена множественного наследования наследованием от интерфейсов в других языках объектно-ориентированного программирования.

**Множественное наследование**

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

class TDelimitedReader : public TTextReader, public TStringList

{

...

};

Объект класса TDelimitedReader содержит все поля и методы базовых классов TTextReader и TStringList. При этом в классе TDelimitedReader можно переопределять виртуальные методы каждого 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.

Неоднозначность при множественном наследовании:

class TTextReader

{

virtual void NextLine();

...

};

class TStringList

{

public:

virtual void NextLine();

...

};

class TDelimitedReader: public TTextReader, public TStringList

{

...

};

TDelimitedReader\*Reader;

Reader->NextLine(); // Ошибка. Неоднозначность.

Неоднозначность возникает потому, что в классе TDelimitedReader существуют две таблицы виртуальных методов и неизвестно, к какой из них надо обращаться за методом NextLine(). Поэтому последний оператор должен быть скорректирован на следующий:

Reader->TTextReader::NextLine();

или:

Reader->TStringList::NextLine();

В С++ для классов поддерживается столько таблиц виртуальных методов, сколько у него базовых классов. При перекрытии общего виртуального метода, существующего в нескольких базовых классах, происходит замещение адреса во всех таблицах виртуальных методов.

Перегрузка функций по типам аргументов не приводит к разрешению неоднозначности.

Если функция NextLine() была объявлена с различной сигнатурой в различных классах, то неоднозначность тоже остается.

В некоторых случаях наличие в базовых классах функций с одинаковыми именами (но различным количеством параметров или различными типами параметров) является преднамеренным решением. Чтобы в производном классе открыть нужную функцию нужного базового класса, применяется оператор **using**:

class TTextReader

{

public:

virtual void NextLine();

...

};

class TStringList

{

public:

virtual void NextLine(int);

...

};

class TDelimitedReader : public TTextReader, public TStringList

{

public:

using TStringList::NextLine;

virtual void NextLine(int);

...

};

**Проблема повторяющихся базовых классов**

Классы TStringList и TTextReader в примере выше могут иметь одинаковый базовый класс, например TObject. Из-за дублирования полей возникает неоднозначность при обращении к полю класса TObject из метода класса TDelimitedReader. Проблема решается с помощью уточненного имени:

TTextReader::m\_Field;

TStringList::m\_Field;

Однако главная проблема состоит в том, что одна сущность дублируется внутри базового класса. Её можно решить, применив виртуальные базовые классы:

class TDelimitedReader: public TTextReader, public TStringList

{

...

};

class TTextReader: public TObject

{

...

};

class TStringList: virtual public TObject

{

...

};

Обычное наследование соответствует агрегации всех полей базового класса. Виртуальное наследование соответствует агрегации ссылки на поля базового класса.

*Что такое агрегация?*

*Например, рассмотрим отношения между человеком и его домашним адресом. У каждого человека есть свой адрес. Однако этот адрес может принадлежать более чем одному человеку в моменте, например, вам и вашему соседу по комнате или родственникам, которые живут вместе с вами. К тому же этот адрес не управляется человеком — адрес существовал до того, как человек заселился и будет существовать после того, как человек выселится. Кроме того, человек знает, по какому адресу он живет, но адрес, в свою очередь, не знает, что это за человек и вообще, сколько их там находится. Такие отношения и являются агрегацией.*

*Это было отступление, дальше обратно к теме:*

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.

<http://citforum.ru/programming/cpp_march/cpp_080.shtml> - норм статья

## 30. Виртуальные методы в языке C++. Недостатки синтаксиса определения и перекрытия виртуальных методов в языке C++. Понятие константного метода. Проблемы, порождаемые наличием константных методов.

(из конспекта лекций)В С++ виртуальные методы определяются при помощи ключевого слова **virtual**:

class TTextReader

{

protected:

virtual void NextLine() = 0;

...

};

При перекрытии виртуального метода ключевое слово **virtual** можно записать, а можно и опустить. Синтаксис перекрытия 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**.

(из интернета)

Виртуальные функции — специальный вид функций-членов класса. Виртуальная функция отличается об обычной функции тем, что для обычной функции связывание вызова функции с ее определением осуществляется на этапе компиляции. Для виртуальных функций это происходит во время выполнения программы.  
Для объявления виртуальной функции используется ключевое слово virtual. Функция-член класса может быть объявлена как виртуальная, если класс, содержащий виртуальную функцию, базовый в иерархии порождения; реализация функции зависит от класса и будет различной в каждом порожденном классе.   
***Виртуальная функция — это функция, которая определяется в базовом классе, а любой порожденный класс может ее переопределить. Виртуальная функция вызывается только через указатель или ссылку на базовый класс.***  
Определение того, какой экземпляр виртуальной функции вызывается по выражению вызова функции, зависит от класса объекта, адресуемого указателем или ссылкой, и осуществляется во время выполнения программы. Этот механизм называется динамическим (поздним) связыванием или разрешением типов во время выполнения.  
Указатель на базовый класс может указывать либо на объект базового класса, либо на объект порожденного класса. Выбор функции-члена зависит от того, на объект какого класса при выполнении программы указывает указатель, но не от типа указателя. При отсутствии члена порожденного класса по умолчанию используется виртуальная функция базового класса.

Если функция отмечена как виртуальная, то все соответствующие переопределения тоже считаются виртуальными, даже если возле них явно не указано ключевое слова virtual. Однако, наличие ключевого слова virtual возле методов дочерних классов послужит полезным напоминанием о том, что эти методы являются виртуальными, а не обычными. Следовательно, полезно указывать ключевое слово virtual возле переопределений в дочерних классах, даже если это не является строго необходимым. **Типы возврата виртуальной функции и её переопределений должны совпадать.**

При создании объекта класса Child сначала создается родительская часть этого объекта, а затем уже дочерняя. Если вы будете вызывать виртуальную функцию из конструктора класса Parent при том, что дочерняя часть создаваемого объекта еще не была создана, то вызвать дочерний метод вместо родительского будет невозможно, так как объект child для работы с методом класса Child еще не будет создан. В таких случаях, в языке C++ будет вызываться родительская версия метода.

Аналогичная проблема существует и с деструкторами. Если вы вызываете виртуальную функцию в теле деструктора класса Parent, то всегда будет вызываться метод класса Parent, так как дочерняя часть объекта уже будет уничтожена.

**Правило: Никогда не вызывайте виртуальные функции в теле конструкторов или деструкторов.**

**Главный недостаток виртуальных методов**: Обработка и выполнение вызова виртуального метода занимает больше времени, чем обработка и выполнение вызова обычного метода. Кроме того, компилятор также должен выделять один дополнительный указатель для каждого объекта класса, который имеет одну или несколько виртуальных функций.

Ссылки с примерами:

<https://ravesli.com/urok-163-virtualnye-funktsii-i-polimorfizm/>

<https://prog-cpp.ru/cpp-virtual/>

**Константные методы**

(конспект лекций)Если в программе объект объявлен с помощью модификатора **const**, то у него можно вызывать лишь те методы, которые объявлены с этим же модификатором:

class TTextReader

{

public:

int ItemCount() const;

...

};

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). Пример:

mutable int m\_RefCount;

Так же решить проблему можно при помощи перегрузки метода класса без модификатора **const**:

class TTextReader

{

public:

int ItemCount() const;

int ItemCount();

int ItemCount() const volatile;

int ItemCount() volatile;

...

};

// volatile str:string s;

// int i = s.length(); - оишбка, тк length дожлен был быть volatile

Варианты объявления:

volatile TTextReader r;

const volatile TTextReader r;

const TTextReader r;

TTextReader r;

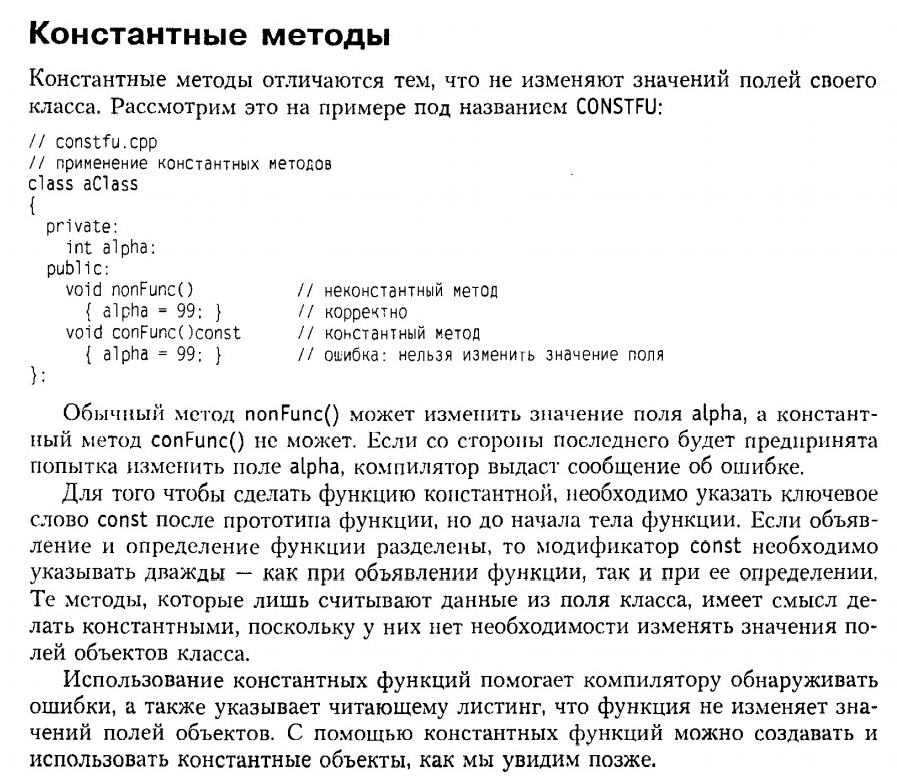
Ключевое слово **volatile** запрещает кэшировать значение переменной. Если в программе происходит считывание значения переменной, объявленной с этим ключевым словом, то значение считывается из памяти, а не из регистров, а запись всегда производится в память, в которой размещается данная переменная.

Если ключевое слово **volatile** не указано, то оптимизатор C++ имеет право выполнять регистровые операции (оптимизации) при чтении и записи переменных, а так же размещать их в регистрах.

Четвертый оператор (**dynamic**\_**cast**) соответствует оператору **as** в Delphi. Для работы этого оператора нужно в опциях компилятора включить опцию RTTI. Если это выполнено, то оператор dynamic\_cast работает, как static\_cast.

Оператор dynamic\_cast работает по-разному в зависимости от того, применяется он к ссылке на объект (&) или указателю на объект (\*). Если оператор применяется к ссылке на объект, то преобразование не может быть выполнено и возникает исключительная ситуация. Если он применяется к указателю на объект и преобразование не может быть выполнено, оператор возвращает NULL.

(это уже из книг и интернета)



(простыми словами со стэковерфлоу)

* Константные методы служат для определения того, что можно сделать с классом без побочного действия на его состояние, и что изменяет его состояние. Пользователю класса, в котором правильно расставлены const, в некоторых случаях, например, удаётся избежать лишнего копирования объектов без потери данных. Также при правильном использовании const на уровне компиляции исключается случайное изменение объекта в методе, который не должен ничего в объекте менять.
* Несколько слов о применении константных методов. Классы с константными методами удобно применять:

1.Когда над проектом работают несколько программистов. Когда проектируется класс разработчик сразу определяет, что и при каких условиях может меняться, а что должно остаться неизменным. Константные классы тут как раз к месту для построения всех "запретительных заборов", чтобы другой разработчик случайно не изменил состояние объекта. Как правило, классы с константными методами не имеют конструктор по умолчанию, а только конструкторы, зависящие от каких-то объектов. Это очень удобно, так как все члены будут инициализированы. Кстати, это единственный способ инициализации константных членов.

2.Использование объектов const. Если создается константный объект, то для обращения к его методам и членам надо использовать константные методы. Например, если в функцию передается указатель на объект, но хочется обезопасить его от изменения, то нужно его объявлять как const

(и ссылка)

<https://ravesli.com/urok-123-klassy-i-const/>

## 31. Операторы приведения типа в языке C++. Операторы приведения типа в языке C++: const\_cast, reinterpret\_cast, static\_cast, dynam-ic\_cast

Существует четыре оператора преобразования типа в С++:

reinterpret\_cast<тип>(переменная)

static\_cast<тип>(переменная)

const\_cast<тип>(переменная)

dynamic\_cast<тип>(переменная)

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);

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 к константным и наоборот:

void f2(char \*s);

void f1(const char \*s)

{

...

f1(const char \*s);

...

f2(const\_cast<char>(s))

...

};

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.

Объявление f(char const \*s) означает, что указатель s изменять нельзя.

Так же можно сделать объявление: f(const char const \*s), которое будет означать, что ни указатель, ни переменную изменять нельзя.

Если в программе объект объявлен с помощью модификатора const, то у него можно вызывать лишь те методы, которые объявлены с этим же модификатором:

class TTextReader

{

public:

int ItemCount() const;

...

};

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). Пример:

mutable int m\_RefCount;

Так же решить проблему можно при помощи перегрузки метода класса без модификатора const:

class TTextReader

{

public:

int ItemCount() const;

int ItemCount();

int ItemCount() const volatile;

int ItemCount() volatile;

...

};

// volatile str:string s;

// int i = s.length(); - оишбка, тк length дожлен был быть volatile

Варианты объявления:

volatile TTextReader r;

const volatile TTextReader r;

const TTextReader r;

TTextReader r;

Ключевое слово volatile запрещает кэшировать значение переменной. Если в программе происходит считывание значения переменной, объявленной с этим ключевым словом, то значение считывается из памяти, а не из регистров, а запись всегда производится в память, в которой размещается данная переменная.

Если ключевое слово volatile не указано, то оптимизатор C++ имеет право выполнять регистровые операции (оптимизации) при чтении и записи переменных, а так же размещать их в регистрах.

dynamic\_cast

Cоответствует оператору as в Delphi. Для работы этого оператора нужно в опциях компилятора включить опцию RTTI. Если это выполнено, то оператор dynamic\_cast работает, как static\_cast.

Оператор dynamic\_cast работает по-разному в зависимости от того, применяется он к ссылке на объект (&) или указателю на объект (\*). Если оператор применяется к ссылке на объект, то преобразование не может быть выполнено и возникает исключительная ситуация. Если он применяется к указателю на объект и преобразование не может быть выполнено, оператор возвращает NULL.

## 33. Перегрузка операторов в языке C++. Перегрузка бинарных операторов. Перегрузка унарных операторов. Перегрузка операторов преобразования типа.

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.

Бинарные операторы

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.

Пример:

class X

{

public:

void operator +(int);

X(int);

};

void operator +(X, X);

void operator +(X, double);

void Proc(X, a)

{

a + 1; // a.operator +(1)

1 + a; // ::operator +(X(1),a)

a + 1.0; // ::operator +(a, 1.0)

}

Унарные операторы

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);

Запрещено перегружать операторы, которые нарушают грамматику языка.

Существует три оператора, которые следует определить внутри класса в виде методов:

operator =

operator []

operator ->

Это гарантирует, что в левой части оператора будет записан lvalue (присваиваемое значение).

Операторы преобразования типов

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:

X::operator T() // определяет преобразования класса X в тип данных T (T — класс или базовый тип)

Пример:

class Number

{

public:

operator int();

operator Complex();

};

Number::operator int()

{

...

}

Number::operator Complex()

{

...

}

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.

Пример:

class File

{

public:

explicit File(const char \*name); // одновременно не могут быть определены, надо выбирать один из них

explicit File(const char \*name, int mode = FILE\_READ);

};

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.

Если объект создается на стеке, то неявное преобразование типа часто бывает необходимо, тогда слово explicit писать надо. Так же его надо писать, когда объект создается динамически.

При перегрузке операторов нужно быть внимательным к типу возвращаемого значения: для некоторых операторов объект возвращается по ссылке, для некоторых — по значению:

X operator; // по значению

X &operator; // по ссылке

Для некоторых операторов возможен и первый и второй вариант перегрузки, поэтому программисту следует определяться с вариантом перегрузки.

Замечание по поводу преобразования типа в тернарном операторе (c ? x : y).

class A { ... };

class B: public A { ... };

class C: public A { ... };

Запись

A\* p = cond ? new B : new C;

вызовет ошибку компилятора, поскольку между типами выражений "new B" и "new C" выбирается общий тип, а такого нет. Ошибку следует устранить, выполнив преобразование "new B" или "new C" к общему типу, например:

A\* p = cond ? (A\*)new B : new C;

или

A\* p = cond ? new B : (A\*)new C;

или

A\* p = cond ? (A\*)new B : (A\*)new C; // самый лучший вариант

## 34. Шаблоны функций и классов в языке C++. Шаблоны функций в языке C++. Шаблоны классов в языке C++. Специализации шаблонов классов. Проблемы шаблонов в языке C++.

Шаблоны

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)

{

...

}

Шаблоны функций

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.

Разработка шаблонов обычно ведется так:

1. разрабатывается класс или функция конкретного типа данных;

2. этот класс или функция параметризуются, то есть создается шаблон.

• существенно замедляется время компиляции. В больших проектах оно может доходить до 30-60 минут.

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

Компиляция относительно большого проекта в отладочном режиме может потребовать до 10 ГБ.

Перегрузка шаблонов функций

Можно объявить несколько шаблонов функций с одним и тем же именем, а так же можно объявить комбинацию шаблонов и обычных функций с одинаковым именем:

template <class T> T sqrt(T x)

template <class T>

complex<T> sqrt(complex<T> x);

template <class T>

double sqrt(double x);

void Proc(complex<double> z)

{

sqrt(2); // sqrt<int>(int)

sqrt(2.0); // sqrt(double)

sqrt(z); // sqrt<double>(comlex<double>)

}

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:

class Allocator

{

... // malloc, free, calloc

};

template <class T, class A = Allocator>

class basic\_string

{

...

};

basic\_string<char, MyAllocator> mystr;

basic\_string<char> commonstr;

Специализации шаблонов

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:

template <class T> template <>

class vector class vector<void\*>

{ {

... ...

}; };

void Proc()

{

vector<int> vi;

vector<void\*> vp;

...

}

Также может применяться частичная специализация шаблонов:

template <class T>

class vector<T\*> : private vector<void\*>

{

...

};

template <>

class vector<void\*>

{

...

};

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.

Использование шаблонов при создании новых типов данных

На основе шаблонов можно создать новый типа данных, использование которого позволит упростить программный код и избежать возможных ошибок.

Существует два способа создания нового типа данных на базе шаблона:

можно воспользоваться оператором typedef:

typedef basic\_string<char> string;

Далее типом данных string можно пользоваться, забыв, что это шаблон.

На самом деле для компилятора оператор typedef является как бы макроподстановкой, то есть при использовании типа string компилятор всегда подставляет значение типа basic\_string<char>.

Если шаблонные классы агрегируют другие шаблонные классы и порождаются от шаблонных классов, то длина программного идентификатора внутри объектного модуля может быть очень велика. Порой она превышает 4 КБ. Некоторые компиляторы на платформе Unix имеют ограничения на длину программного идентификатора внутри объектного модуля. Поэтому при использовании шаблонов (в особенности библиотеки stl) нередко возникает проблема с компиляцией.

Например, выражение:

string::iterator;

на самом деле в объектном модуле выглядит так:

basic\_string<char>::iterator;

Оператор typedef по существу отличается от раздела type в Delphi. В С++ упрощение записи — макроподстановка.

• наследование

class string : public basic\_string<char>

{

...

};

При наследовании создается именно новый тип данных, поэтому в объектный модуль в данном случае помещается идентификатор следующего вида:

string::iterator;

Таким образом, использование наследования позволяет решить проблему длины идентификаторов при работе с шаблонами.

Стандартная библиотека шаблонов

Перечислим, что содержится в стандартной библиотеке шаблонов:

• Классы и шаблоны для организации потоков ввода/вывода

В языке С++ вместо функций printf и scanf предлагается использовать объекты потоковых классов:

std::cout;

std::cin;

Вывод осуществляется с помощью оператора сдвига:

std::cout << "Hello!";

int n;

std::cin >> n;

Чтобы перевести строку надо сделать следующую запись:

std::cout << "Hello!" << std::endl; // или "\n"

Объекты cout и cin являются экземплярами классов ostream и istream. Существуют также классы iostream (класс для ввода/вывода) и streambuf (позволяет выполнить буферизованный ввод/вывод).

В программе не следует смешивать потоковый ввод/вывод с функциями printf и scanf. Если все же это происходит, То между блоками кода, использующими тот или иной подход, надо выполнять вызов функции fflush — сброс буферов.

• Ссылки

Для работы с ссылками подключается файл:

#include <string>

Расширение .h у файлов стандартной библиотеки отсутствует.

Среди строк наибольшей популярностью пользуются следующие классы:

std::string

std::wstring

• Контейнеры

В языке С++ существуют следующие контейнеры:

std::vector<T> // обычный массив

std::list<T> // список, реализация не уточняется.

std::queve<T> // FIFO

std::deque<T> // двунаправленная очередь

std::stack<T> // LIFO

std::map<K,T> // список объектов T, индексированных ключами K

std::set<T> // множество

std::bitset // массив битов

Для использования любого из шаблонов надо подключить заголовочный файл, название которого совпадает с названием подключаемого шаблона.

В примере выше кроме параметра <T> есть еще, как правило, три параметра. Здесь они преднамеренно опущены и для них заданы стандартные значения.

Например, среди этих параметров есть allocator — это класс, который отвечает за создание или удаление элементов контейнера. Благодаря нему, можно создавать элементы в любой области памяти.

• Итераторы

#include <iterator>

Итератор — абстракция указателя на элемент контейнера.

Пример использования:

#include <iterator>

void strlen(const cahr \*str)

{

const char \*p = str;

while(\*p != 0)

{

...

++p;

}

...

}

Указатель в строке — это итератор по строке.

Можно сказать, что в примере выше типы данных char\* и const char\* являются итераторами строки (обычной 0-терминированной).

С помощью оператора typedef можно определить такой тип данных и затем их использовать:

typedef char \*iterator;

typedef const char \*const\_iterator;

Внутри каждого контейнера стандартной библиотеки С++ определены два типа данных: iterator и const\_iterator, которые фактически являются указателями на элемент контейнера.

Работа с этими типами данных происходит следующим образом:

void Find(std::vector<MyObject\*> &v, const char \*s)

{

typename std::vector<MyObject\*>::iterator it = v.begin();

while (it != v.end())

{

const char \*szName = (\*it).Name;

if (strcmp(szName, s) == 0)

return true;

++it;

}

return false;

}

В стандартном контейнере существуют функции begin() и end(), которые возвращают соответственно итераторы на первый и последний элементы контейнера. Физически функция end() возвращает NULL.

Если итератор адресует объект, то доступ к полям следует осуществлять с помощью оператора \*. Допустимо использование и оператора ->, но он может быть переопределен и поэтому работа операторов \*. и -> может отличаться. Переход к следующему элементу контейнера выполняется префиксным инкрементом ++it. Допустимо использование постфиксного оператора it++, но в последнем случае может возникнуть неоднозначность в том, что увеличивается на единицу — итератор или значение, возвращаемое итератором.

• Алгоритмы

#include <algorithm>

#include <cstdlib>

В стандартной библиотеке существует несколько стандартных алгоритмов, которые следует изучить до того, как разрабатывать собственные алгоритмы над контейнерами.

find()

sort()

\*bsearch() // бинарный поиск

\*qsort() // быстрая сорировка

Недостатком bsearch является то, что не возвращается место вставки элемента.

• Утилиты

#include <utility>

#include <functional>

#include <memory>

#include <ctime>

В utility переопределен шаблон pair<F,S>.

В fuctional определены объекты функций — это такие объекты, которые могут использоваться как функции. Объекты функций применяются в алгоритмах. Например, выполняется обход контейнера и вызывается некоторая функция для каждого элемента контейнера. Вместо функции можно подставить объект-функцию. В этом и состоит смысл объекта-функции.

memory — распределение памяти для контейнера (здесь находятся стандартные алгоритмы).

В ctime находятся функции времени и даты.

• Диагностика

#include <stdexcept> // классы исключительных ситуаций

#include <cassert> // макрос assert. Вместо него лучше использовать

// исключительные ситуации

#include <cerrno> // файлы обработки ошибок в стиле С, то есть функции,

// которые возвращают коды ошибок

• Локализация

Поддержка различных языков:

#include <locale>

#include <clocale>

• Поддержка языка программирования С++

Крайние граничные значения типов данных:

#include <limits>

#include <climits>

#include <typeinfo> // поддержка RTTI

#include <exception> // поддержка исключительных ситуаций

• Работа с числами

#include <complex>

#include <volarray>

#include <numeric>

#include <cmath>

#include <cstdlib>

## 35. Сравнение языков объектно-ориентированного программирования: C#, Java, C++.

Сравнение C# и Java

<https://ru.wikipedia.org/wiki/Сравнение_C_Sharp_и_Java>

Сравнение C++ и Java

<https://blog.ithillel.ua/articles/raznica-mezhdu-yazykami-programmirovaniya-c-i-java>

<http://www.interface.ru/home.asp?artId=3567>

Сравнение C# и C++

<http://programming-lang.com/ru/comp_programming/robinson/0/j696.html>

Сравнение C#, Java и C++

<http://msnet.narod.ru/art/art_001/art_001.htm>

<http://zei.narod.ru/Comparison_C__Java_Cpp_3.pdf>