1. **Основные подходы к разработке компьютерных программ (процедурно-ориентированное, структурное, модульное, объектно-ориентированное программирование). Основные концепции объектно-ориентированного программирования. Этапы объектно-ориентированного программирования. Достоинства и недостатки объектно-ориентированного программирования.**

**Основные подходы к разработке компьютерных программ**

*Процедурное программирование* — программирование, при котором последовательно выполняемые операторы можно собрать в подпрограммы, то есть более крупные целостные единицы кода, с помощью механизмов самого языка.

*Структурное программирование* — парадигма программирования, в основе которой лежит представление программы в виде иерархической структуры блоков

* любая программа строится из трёх базовых управляющих конструкций (последовательность, ветвление, цикл)
* конструкции могут быть вложены друг в друга
* повторяющиеся фрагменты можно оформить в виде процедур и функций
* конструкций один вход один выход
* разработка пошагово, методом «сверху вниз»

*Модульное программирование* — программа разбивается на группу компонентов, называемых модулями, причем каждый из них имеет свой контролируемый размер, четкое назначение и детально проработанный интерфейс с внешней средой.

*Объектно-ориентированное программировани*е — представление программы в виде совокупности взаимодействующих объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию наследования.

**Основные концепции объектно-ориентированного программирования**

* Инкапсуляция. Это сокрытие информации и комбинирование данных и функций (методов) внутри объекта.
* Наследование. Построение иерархии порожденных объектов с возможностью для каждого такого объекта-наследника доступа к коду и данным всех порождающих объектов-предков.
* Полиморфизм — присваивание действию одного имени, которое затем разделяется вверх и вниз по иерархии объектов, причем каждый объект иерархии выполняет это действие способом, подходящим именно ему.
* Абстракция. Моделирование требуемых атрибутов и взаимодействий сущностей в виде классов для определения абстрактного представления системы.

**Этапы объектно-ориентированного программирования**

* Анализ
* Проектирование
* Эволюция
* Модификация.

**Плюсы**

* Уменьшение сложности и повышение надежности программного обеспечения
* Возможность модификации отдельных компонентов программного обеспечения без изменения остальных
* Повторное использование отдельных компонентов программного обеспечения

**Минусы**

* снижение производительности функционирования ПО
* высокие начальные затраты.
* Объектная декомпозиция существенно отличается от функциональной(психологические трудности дополнительные финансовые затраты.(???)

1. **Общая характеристика языка С++. Описание класса в языке С++. Создание экземпляра класса. Указатель this. Статические члены класса. Дружественные классы и методы.**

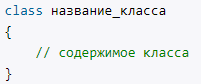
**Общая характеристика языка С++**

C++ - компилируемый статически типизированный язык программирования общего назначения. ИСПОЛЬЗУЕТ ВСЕ ЧТО ВЫШЕ

**Описание класса в языке С++**

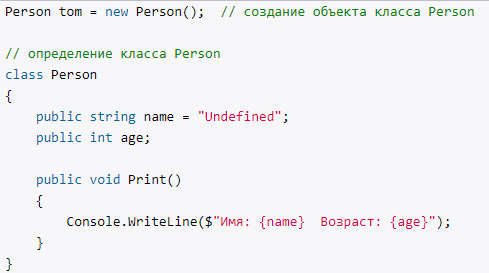
Класс — это тип данных, определяемый пользователем, который похож на структуру или запись, так как содержит поля с данными, но кроме полей в описание класса входят также функции, которые предназначены для обработки полей. Такие функции называются методами класса. Поля и методы называются элементами класса.

Экземпляр класса является объектом данного класса и через него можно с ним взаимодействовать.



**Создание экземпляра класса**

Конструктор по умолчанию, а также можно через параметры и с помощью конструктора копирования



**Указатель this**

*Указатель this* — это указатель на адрес объекта класса, при этом он является скрытым первым параметром любого метода класса (кроме статических методов), а типом указателя выступает имя класса.

**Статические члены класса**

Статический метод (static) класса – метод, не имеющий доступа к состоянию (полям) объекта, то есть к переменной this. Аналогично тому, как статический метод не привязан к объекту, статическое поле не является частью никакого объекта, оно будет создаваться в одном экземпляре и будет доступно из любого объекта класса или через имя класса. Статические свойства класса используется для того, чтобы взаимодействовать с ними без создания нового контекста или экземпляра класса. Хранятся (.data или .bss)

**Дружественные классы и методы**

Дружественные функции - это функции, которые не являются членами класса, однако имеют доступ к его закрытым членам - переменным и функциям, которые имеют спецификатор private. Для определения дружественных функций используется ключевое слово friend.

1. **Инкапсуляция. Преимущества и недостатки инкапсуляции. Область видимости элементов класса (private, public, protected). Внутренняя и внешняя реализация методов класса. Организация исходных файлов (файл заголовков и файл реализации).**

**Инкапсуляция**

Инкапсуляция — это свойство, позволяющее объединить в классе и данные, и методы, работающие с ними и скрыть детали реализации от пользователя.

Преимущества

* Полный контроль над входящими и исходящими данными.
* Можно править реализацию методов компонентов.
* меняется только реализация логики, а не сам интерфейс

Недостатки

* ошибка в библиотеке, которую используете, то Вам будет трудно ее исправить (????не понял)
* Снижается скорость работы приложения
* Расширяемость поведения объекта падает до нуля (????)

**Область видимости**

**public** делает свойство/метод доступным из любого места, других классов и экземпляров объекта.

**private** делает свойство/метод видимым только в его собственном классе.

**protected** делает свойство/метод видимым во всех классах, расширяющих текущий класс, включая родительский класс.

**Внутренняя и внешняя реализация методов класса**

Внутреннее программирование класса – это разработка программного кода, который находится в теле разрабатываемого класса (точнее, в теле его методов).

Внешнее программирование – это разработка кода вне тела проектируемого класса, который создает объекты класса, работает с данными этих объектов и вызывает методы.

**Организация исходных файлов**

Заголовочные файлы содержат только описания. Прежде всего, это прототипы функций. Прототип функции описывает имя функции, тип возвращаемого значения, число и типы ее аргументов

Заголовочный файл представляет собой файл, содержащий объявления C и определения макросов для совместного использования нескольких исходных файлов(#include)

* Файлы заголовков системы объявляют интерфейсы к частям операционной системы
* Собственные заголовочные файлы содержат декларации для интерфейсов между исходными файлами вашей программы

Файлы реализации — это отдельные модули, которые разрабатываются и транслируются независимо друг от друга и объединяются при создании выполняемой программы.

Файлы реализации могут подключать описания, содержащиеся в заголовочных файлах. Сами заголовочные файлы также могут использовать другие заголовочные файлы.

1. **Конструкторы и деструкторы. Порядок вызова конструкторов и деструкторов. Конструктор по умолчанию, конструктор с параметрами. Конструктор-копировщик, конструктор преобразования.**

**Конструкторы и деструкторы**

Конструкторы — это особые методы, которые выполняются автоматически при создании объекта. При этом конструктор имеет то же самое имя, что и класс.

Деструкторы — это особые методы, которые выполняются автоматически при уничтожении объекта. Деструкторы имеют то же самое имя, но со знаком ~. Деструктор вызывается в том случае, когда программа покидает область видимости, после этого происходит чистка полей класса.

**Порядок вызова конструкторов и деструкторов**

Конструкторы вызываются в порядке вниз по иерархии наследования классов, т.е. начиная с самого базового класса и заканчивая производным классом.

Деструкторы вызываются соответственно в обратном порядке.

**Конструкторы**

* конструктор по умолчанию (без параметров) – присваивает полям нулевые значения или какие-либо значения по умолчанию;
* конструктор с параметрами — присваивает полям заданные значения – параметры;
* конструктор-копировщик — используется для создания копии объекта. Конструктор-копировщик чаще всего вызывают при передаче в функцию параметра значения, который является объектом.
* Конструктор преобразования вызывается для неявного преобразования. Конструктор-преобразования вызывается автоматически для преобразования значений других типов в объекты данного класса.

Деструктор вызывается в том случае, когда программа покидает область видимости, после этого происходит чистка полей класса.

Деструкторы используются:

1. Деструктор – метод, который вызывается при

удалении экземпляра класса.

В большинстве случае применяется для очистки памяти.

2. Деструктор позволяет инкапсулировать логику освобождения ресурсов.

3. Уведомить о том, что объект был удален (например, для проверки работы сборщика мусора)

4. Для определения виртуального деструктора в абстрактном классе, что позволяет удалять без утечек памяти, имея лишь указатель на базовый класс

5. Можно сохранить важные данный в файл в деструкторе

6. закрытие ресурсов(файлов)

**5. Наследование. Производные классы С++. Преимущества наследования. Конструкторы производных классов. Доступ к наследуемым переменным.**

Наследование - на основе одних классов можно создавать новые классы, при этом новый класс называется - производным или наследником, а класс основа — базовым или предком.

Преимущества:

- эффективное построение сложных иерархий классов с возможностью их удобной модификации.

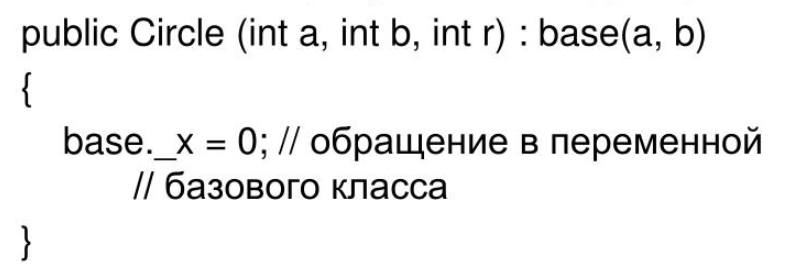
- удобство в сопровождении (дополнении) программного кода путем введения новых классов с новыми возможностями;

- уменьшение количества логических ошибок при разработке сложных программных систем.

- возможность реализовывать известные шаблоны проектирования для построения гибкого кода, который не изменяет предыдущих разработок;

- использование преимуществ полиморфизма невозможно без наследования.

Поскольку конструкторы не наследуются, при создании конструктор производного класса может вызывать на выполнение конструктор базового класса. Конструктор производного класса вызывается по окончании работы конструкторов базовых классов.



**6. Полиморфизм. Применение виртуальных функций.**

К механизму виртуальных функций обращаются в тех случаях, когда в каждом производном классе требуется некоторая компонентная функция, такие функции называются полиморфными.

Виртуальные функции - это функции, объявленные в базовом классе и переопределенные в производных классах.

Они позволяют создавать общий код, который может работать как с объектами базового класса, так и с объектами любого его класса-наследника. При этом базовый класс определяет способ работы с объектами, и любые его наследники могут предоставлять конкретную реализацию этого способа.

**7. Перегрузка операторов. Перегрузка оператора присваивания.**

Перегрузка операторов - возможность существования различных операторов, имеющих одно и то же имя, но различающихся типами параметров, к которым они применяются.



Операция присваивания определена в любом классе, как поэлементное копирование. Эта операция вызывается тогда, когда одному объекту присваивается значение другого объекта.

**8. Использование шаблонов в С++. Определение шаблонов функций и классов. Создание объектов по шаблонам.**

Шаблоны – это функция языка программирования C++, которая позволяет функциям и классам работать с универсальными типами. Это позволяет функции или классу работать со многими различными типами данных, не переписываясь для каждого из них.

**9.Язык Java. Краткая история и характеристика языка. Типы данных и операторы в Java. Классы в Java.**

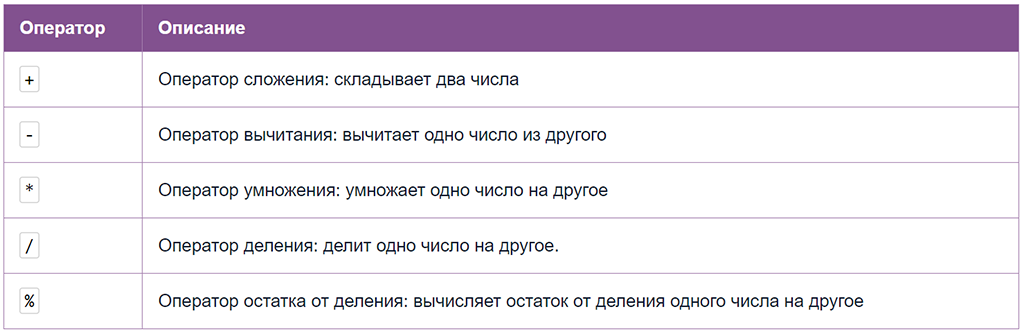
Java - один из самых популярных языков программирования во всем мире. Он был создан Джеймсом Гослингом и Патриком Нотоном, сотрудниками Sun Microsystems, при поддержке Билла Джоя, соучредителя Sun Microsystems. Sun официально представила язык Java на выставке SunWorld 23 мая 1995 года. Затем, в 2009 году, компания Oracle купила компанию Sun, что объясняет, почему язык теперь принадлежит Oracle

Хотя Kotlin является предпочтительным языком для разработки Android в 2021 году, Java по-прежнему остается языком по умолчанию для разработки приложений для Android.Основной причиной этого является тесная интеграция Android с Java. Фактически, части этой ОС, такие как пользовательский интерфейс и определенные основные библиотеки, написаны на Java.

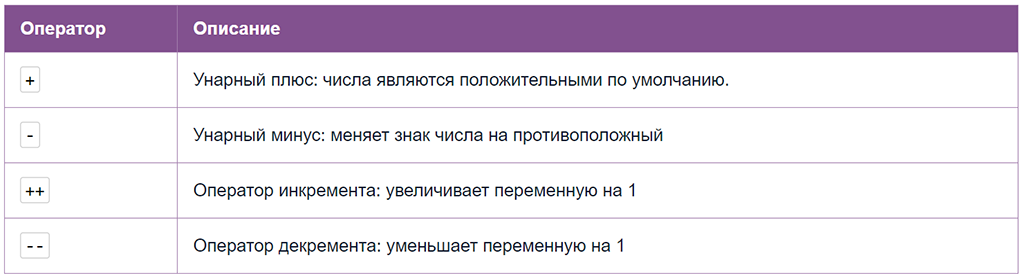
Таблица 2.1. Простые типы, встроенные в Java

| **Тип** | **Описание** |
| --- | --- |
| boolean | Представляет логические значения true и false |
| byte | 8-разрядное целое число |
| char | Символ |
| double | Числовое значение с плавающей точкой двойной точности |
| float | Числовое значение с плавающей точкой одинарной точности |
| int | Целое число |
| long | Длинное целое число |
| short | Короткое число |

*Таблица 1. Бинарные арифметические операторы*



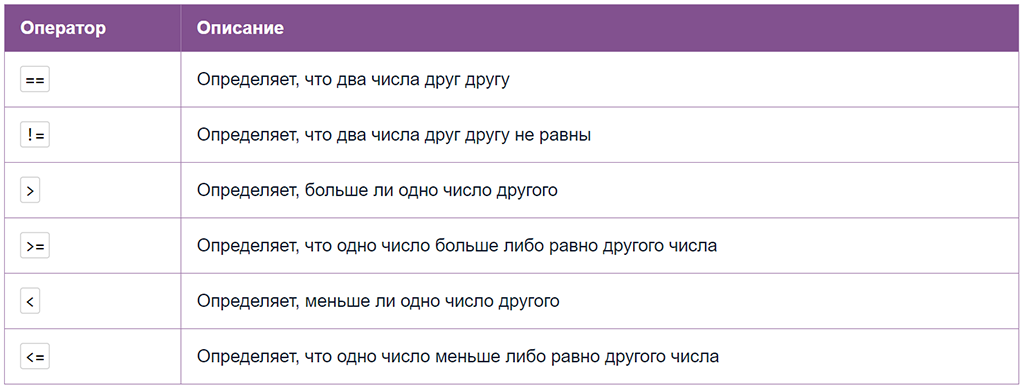
*Таблица 2. Унарные арифметические операторы:*



*Таблица 3. Операторы инкремента-декремента:*



*Таблица 4. Операторы сравнения*



Java является объектно-ориентированным языком, поэтому такие понятия как "класс" и "объект" играют в нем ключевую роль. Любую программу на Java можно представить как набор взаимодействующих между собой объектов.

Шаблоном или описанием объекта является **класс**, а объект представляет экземпляр этого класса. Можно еще провести следующую аналогию. У нас у всех есть некоторое представление о человеке - наличие двух рук, двух ног, головы, туловища и т.д. Есть некоторый шаблон - этот шаблон можно назвать классом. Реально же существующий человек (фактически экземпляр данного класса) является объектом этого класса.

Класс определяется с помощью ключевого слова **сlass**:

|  |  |
| --- | --- |
| 1  2  3 | class Person{    } |

В данном случае класс называется Person. После названия класса идут фигурные скобки, между которыми помещается тело класса - то есть его поля и методы.

Любой объект может обладать двумя основными характеристиками: состояние - некоторые данные, которые хранит объект, и поведение - действия, которые может совершать объект.

Для хранения состояния объекта в классе применяются поля или переменные класса. Для определения поведения объекта в классе применяются методы. Например, класс Person, который представляет человека, мог бы иметь следующее определение:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | class Person{        String name;        // имя      int age;            // возраст      void displayInfo(){          System.out.printf("Name: %s \tAge: %d\n", name, age);      }  } |

В классе Person определены два поля: name представляет имя человека, а age - его возраст. И также определен метод displayInfo, который ничего не возвращает и просто выводит эти данные на консоль.

Теперь используем данный класс. Для этого определим следующую программу:

Как правило, классы определяются в разных файлах. В данном случае для простоты мы определяем два класса в одном файле. Стоит отметить, что в этом случае только один класс может иметь модификатор public (в данном случае это класс Program), а сам файл кода должен называться по имени этого класса, то есть в данном случае файл должен называться Program.java.

Класс представляет новый тип, поэтому мы можем определять переменные, которые представляют данный тип. Так, здесь в методе main определена переменная tom, которая представляет класс Person. Но пока эта переменная не указывает ни на какой объект и по умолчанию она имеет значение **null**. По большому счету мы ее пока не можем использовать, поэтому вначале необходимо создать объект класса Person.

**10.Интерфейсы и пакеты в языке Java. Применение интерфейсов и пакетов в Java.**

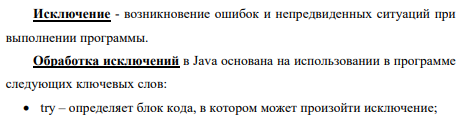
**Изображение выглядит как текст

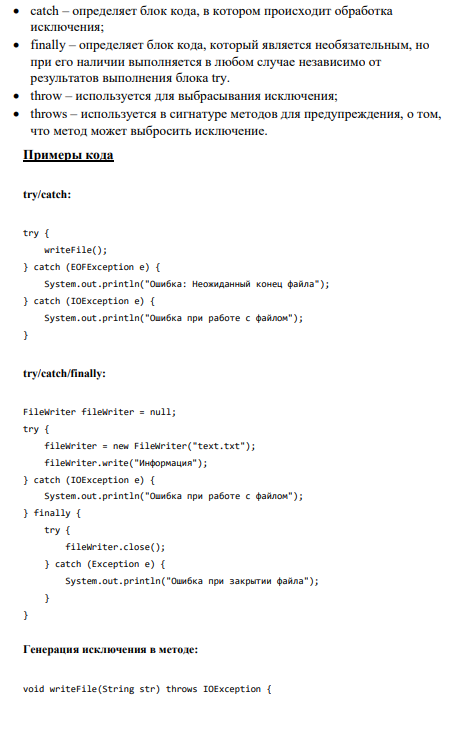
Автоматически созданное описание**

**Изображение выглядит как текст

Автоматически созданное описание**

**11.Обработка исключений в языке Java. Создание собственных классов исключений.**

****

****

**Изображение выглядит как текст

Автоматически созданное описание**

**12. Динамическое определение типов (RTTI). Необходимость в динамическом определении типов. Объект Class. Применение метакласса для получения динамической информации об объекте.**

**Изображение выглядит как текст

Автоматически созданное описание**

**Изображение выглядит как текст

Автоматически созданное описание**

**Изображение выглядит как текст

Автоматически созданное описание**

**13. Контейнеры в языке Java. Использование итераторов.**

**Изображение выглядит как текст

Автоматически созданное описание**

**Изображение выглядит как текст

Автоматически созданное описание**

**14. Потоки в языке Java. Сериализация объектов.**

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

15. Язык C#. Краткая история и характеристика языка. Типы данных и операторы в C#. Классы в C#. Пространства имен в C#.

С# – это объектно-ориентированный язык программирования. Он был создан в период с 1998 по 2002 год командой инженеров Microsoft под руководством Андерса Хейлсберга и Скотта Вильтаумота. Язык входит в семью С-подобных языков. Синтаксис приближен к Java и C++.

Его особенности:

• статистическая типизация,

• поддерживается полиморфизм,

• поддерживается перегрузка операторов,

• доступна делегация, атрибуты, события, обобщенные типы и анонимные функции.

**В** **языке** **C#** реализованы следующие базовые **типы** **данных**:

1. Целые типы: sbyte, byte, short, ushort, int, uint, long, ulong
2. Типы, в которых сохраняются данные с плавающей запятой (соответствуют вещественному типу): float, double, decimal
3. Символьный тип: char
4. Логический тип: bool
5. Строчный тип: string

**Операторы** — это символы, которые используются для выполнения операций с операндами. Операндами могут быть переменные и/или константы.

Операторы используются для управления переменными и их значениями в программе. C# поддерживает ряд операторов, разделенных по типам выполняемых ими операций.

Основной оператор присваивания = используется для присвоения значений переменным.

Арифметические операторы используются для выполнения арифметических операций: сложение, вычитание, умножение, деление и т. д.

Операторы сравнения (они же — операторы отношения) используются для проверки связи между двумя операндами. Если сравнение истинно, результат будет true, иначе — false. Операторы отношения обычно используются в условиях и циклах.  
 

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Операторы инкремента | ++ | и декремента | -- | могут использоваться как |
| префикс и постфикс. Если использовать их как префикс, изменение значения | | | | |

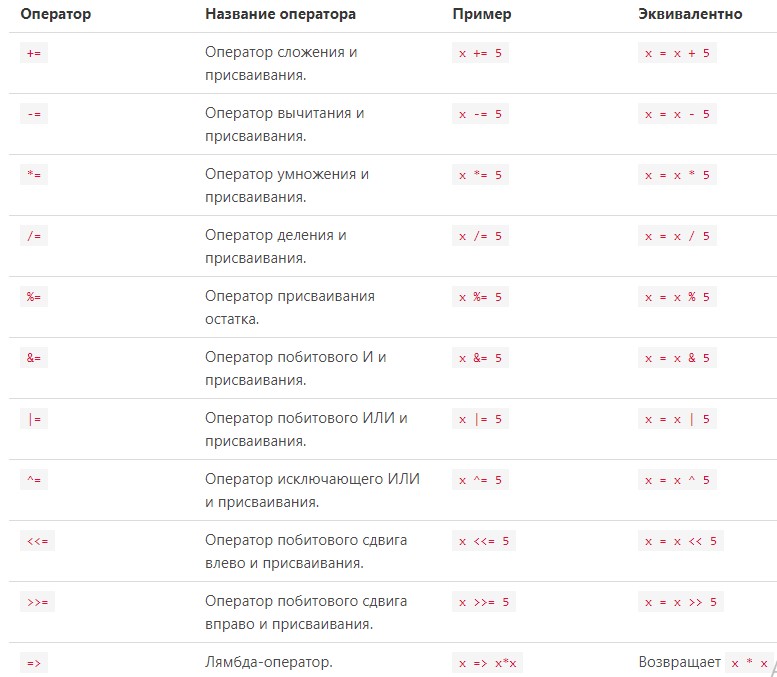
Логические операторы используются для выполнения таких логических операций, как AND (ИЛИ), OR (И) и т. д. Логические операторы работают с булевыми выражениями и возвращают булевые значения true и false. Логические операторы нужны для принятия решений и в циклах. Унарные операторы работают только с одним операндом — это отличает их от всех остальных операторов.   


переменной отображается в той же строке, а если как постфикс, изменение значения переменной отображается в следующей строке.

Побитовые операторы и операторы побитового сдвига используются для выполнения манипуляций с битами.



Составные операторы присваивания



**Класс** — это логическая структура, позволяющая создавать свои собственные пользовательские типы. Класс определяет данные и поведение типа.Классы объявляются с помощью ключевого слова class.

Внутри себя, класс может содержать **методы**, **поля** и **свойства**.

**Методы** похожи на функции из языков группы структурного программирования. Фактически они определяют то, как можно работать с данным классом или объектами класса. **Поля** – это переменные, связанные с данным классом, а **свойства** – это конструкции специального вида, которые упрощают работу с полями

**Пространства имен в C#** определяет область кода, в которой гарантируется уникальность всех идентификаторов. Это своеобразные контейнеры, в которых находятся другие пространства имен, классы и т. д.

В C# определить пространство имен можно с помощью ключевого слова namespace.

*namespace имя\_пространства\_имен*

*{*

*// Тело пространства имен*

*}*

Например: *namespace MyNamespace*

*{*

*class MyClass*

*{*

*public void MyMethod()*

*{*

*System.Console.WriteLine("Создаем пространство имен");*

*}*

*}*

*}*

В приведенном выше примере мы создали пространство имен с именем MyNamespace. Оно состоит из класса MyClass. MyMethod в свою очередь — метод класса MyClass.

16. Свойства и индексаторы в языке C#. Преимущества и недостатки данных элементов языка. Унифицированная система типов в языке C#.

Свойства обеспечивают простой доступ к полям классов и структур, узнать их значение или выполнить их установку.

|  |
| --- |
| [модификаторы] тип\_свойства название\_свойства  {      get { действия, выполняемые при получении значения свойства}      set { действия, выполняемые при установке значения свойства}  } |

Вначале определения свойства могут идти различные модификаторы, в частности, модификаторы доступа. Затем указывается тип свойства, после которого идет название свойства. Полное определение свойства содержит два блока: **get** и **set**.

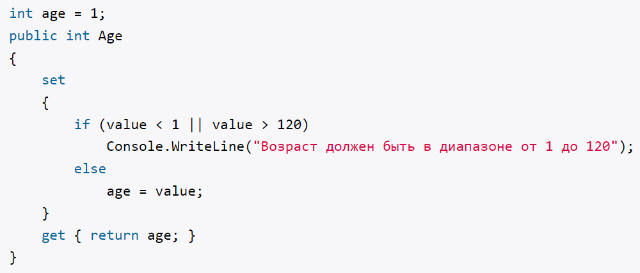
В блоке **get** выполняются действия по получению значения свойства. В этом блоке с помощью оператора **return** возвращаем некоторое значение.

В блоке **set** устанавливается значение свойства. В этом блоке с помощью параметра **value** мы можем получить значение, которое передано свойству.

В программе мы можем обращаться к этому свойству, как к обычному полю. Если мы ему присваиваем какое-нибудь значение, то срабатывает блок **set**, а передаваемое значение передается в параметр **value**.

Если мы получаем значение свойства, то срабатывает блок **get**, который по сути возвращает значение переменной.

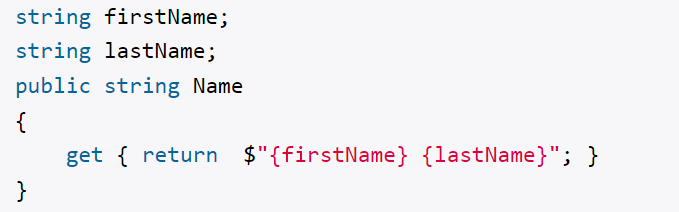
Свойства позволяют вложить дополнительную логику, которая может быть необходима при установке или получении значения.



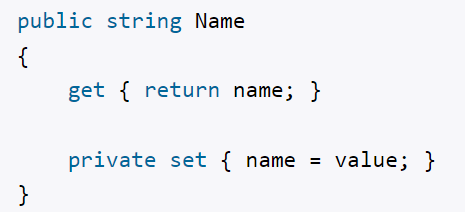
Блоки set и get не обязательно одновременно должны присутствовать в свойстве. Если свойство определяет только блок **get**, то такое свойство доступно только **для чтения** - мы можем получить его значение, но не установить.

И, наоборот, если свойство имеет только блок **set**, тогда это свойство доступно только для записи - можно только установить значение, но нельзя получить:

Свойства необзательно связаны с определенной переменной. Они могут вычисляться на основе различных выражений



Мы можем применять модификаторы доступа не только ко всему свойству, но и к отдельным блокам get и set.



**Индексаторы** позволяют индексировать объекты и обращаться к данным по индексу. Фактически с помощью индексаторов мы можем работать с объектами как с массивами. По форме они напоминают свойства со стандартными блоками get и set, которые возвращают и присваивают значение.

возвращаемый\_тип this [Тип параметр1, ...]

{

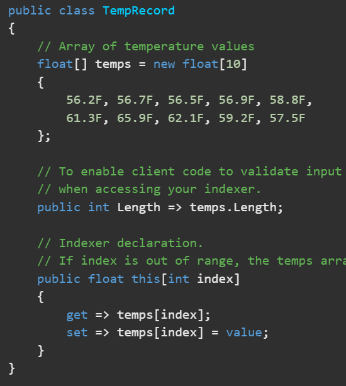
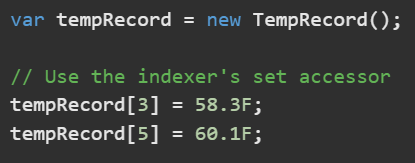
    get { ... }

    set { ... }

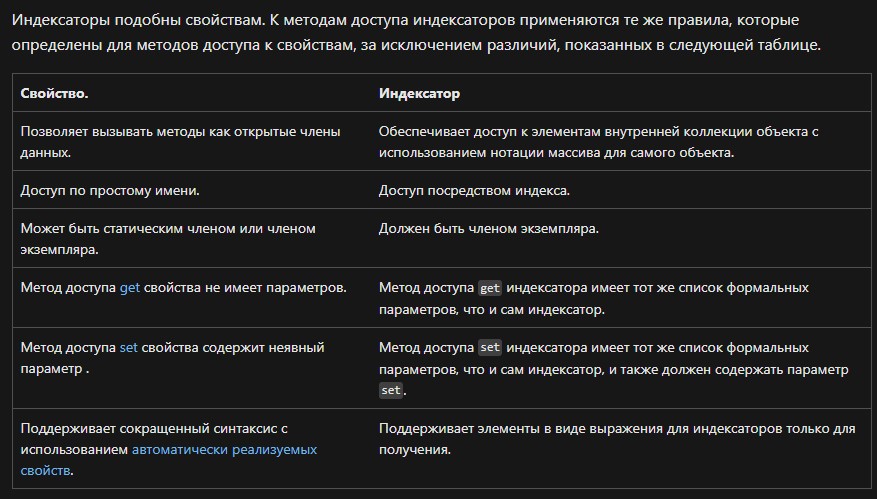
}

В отличие от свойств индексатор не имеет названия. Вместо него указывается ключевое слово **this**, после которого в квадратных скобках идут параметры. Индексатор должен иметь как минимум один параметр.

Применение индексаторов упрощает работу с синтаксисом, позволяя создавать классы, структуры и интерфейсы, к которым клиентские приложения могут обращаться так же, как к массиву. В этом случае компилятор создает свойство Item (или свойство с другим именем, если присутствует IndexerNameAttribute) и соответствующие методы доступа. Индексаторы чаще всего реализуются в типах, предназначенных преимущественно для инкапсуляции внутренней коллекции или массива. Допустим, у вас есть класс TempRecord, представляющий журнал с 10 измерениями температуры по шкале Фаренгейта за период в 24 часа. Этот класс содержит массив temps типа float[] для хранения значений температуры. Реализация индексатора в этом классе позволит клиентам получать доступ к значениям температуры в экземпляре TempRecord, используя float temp = tempRecord[4] вместо float temp = tempRecord.temps[4].

Некоторые средства языка С# являются избыточными - в том смысле, что не добавляют к языку никаких функциональных возможностей, а лишь позволяют в иной форме записать то, что и так может быть выражено достаточно просто. К числу таких средств относятся свойства (properties) и индексаторы (indexers). Такие конструкции языка С#, как свойства, индексаторы, приводят к ситуациям, когда по тексту программы (программной единицы) невозможно понять природу используемых в ней объектов. Свойства могут быть неотличимы от полей, индексируемые объекты - от массивов.



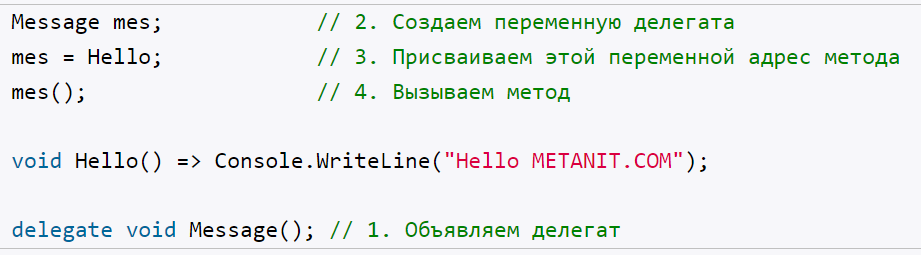
**Унифицированная система типов C#** фактически означает, что типы значений обрабатываются как object ссылки "по запросу". Из-за унификации библиотеки общего назначения, использующие тип object , могут использоваться со всеми типами, производными от object , включая ссылочные типы и типы значений.

17. Делегаты в языке C#. Модель событий в языке С#.

Делегаты представляют такие объекты, которые указывают на методы. То есть делегаты - это указатели на методы и с помощью делегатов мы можем вызвать данные методы.

Для объявления делегата используется ключевое слово **delegate**, после которого идет возвращаемый тип, название и параметры.

delegate void Message();

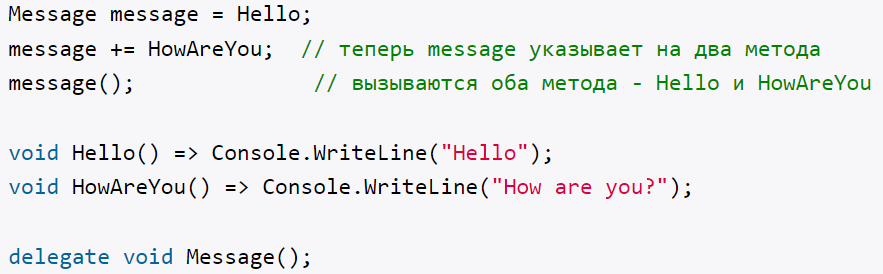


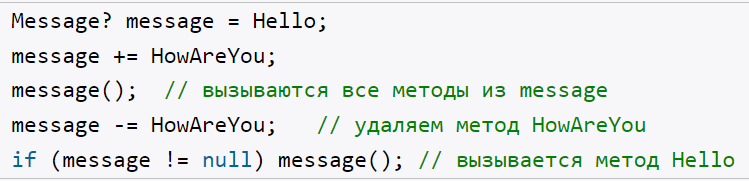
Делегаты могут принимать параметры и возвращать значения.

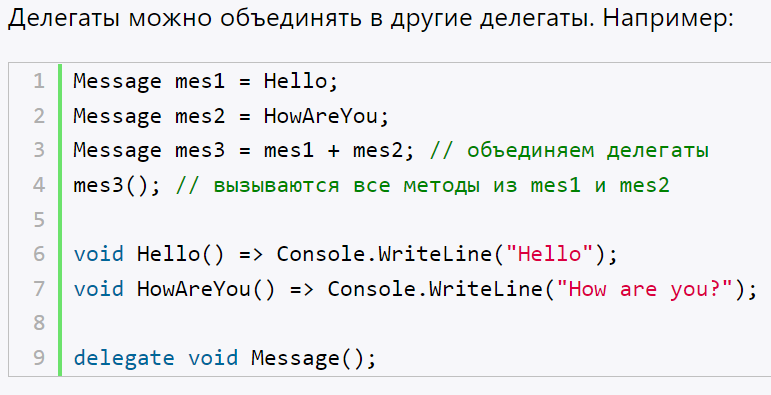
delegate int Operation(int x, int y);

Методы соответствуют делегату, если они имеют один и тот же возвращаемый тип и один и тот же набор параметров.

Для добавления методов в делегат применяется операция **+=**



Подобным образом мы можем удалять методы из делегата с помощью операций **-=**. 



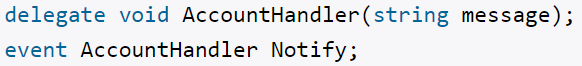
Другой способ вызова делегата представляет метод **Invoke()**:

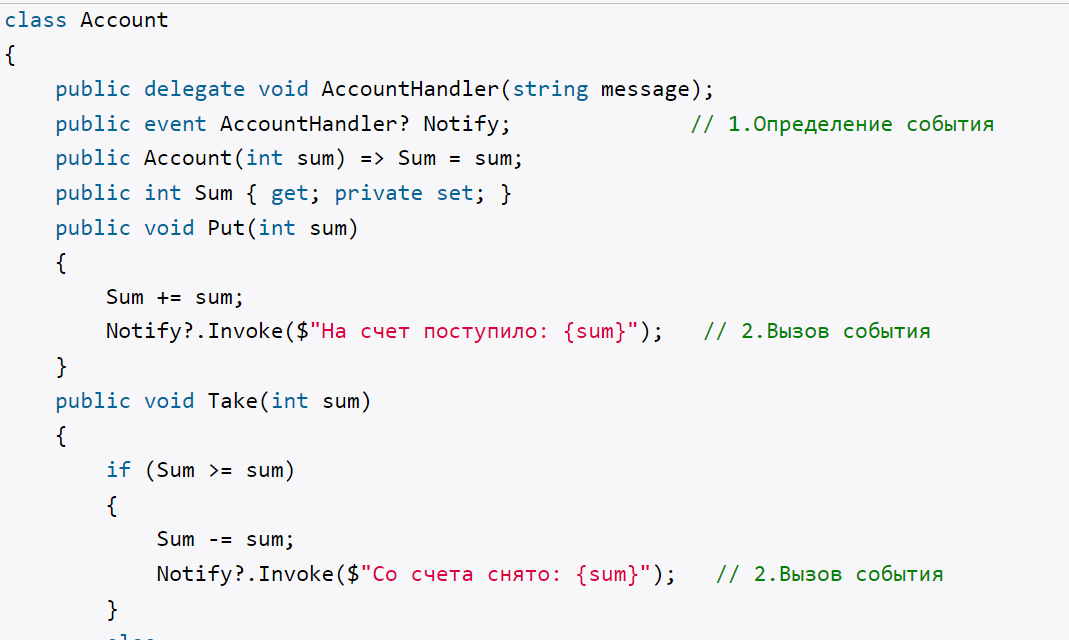
mes.Invoke();

Делегаты могут быть параметрами методов. Также делегаты можно возвращать из методов.

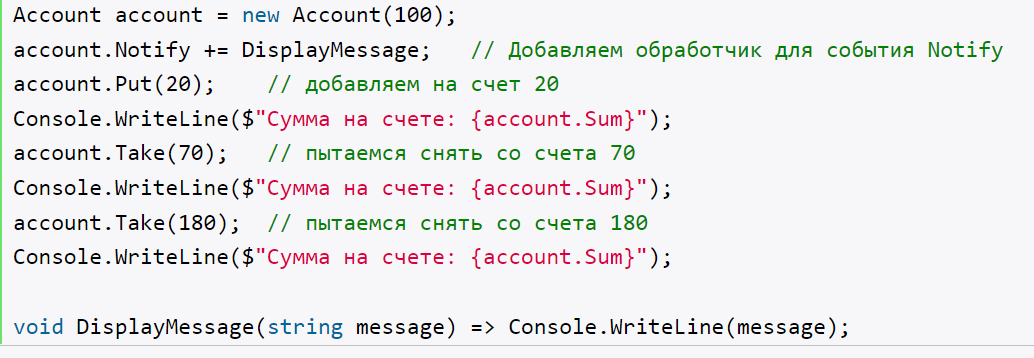
**События** сигнализируют системе о том, что произошло определенное действие.

События объявляются в классе с помощью ключевого слова **event**, после которого указывается тип делегата, который представляет событие.

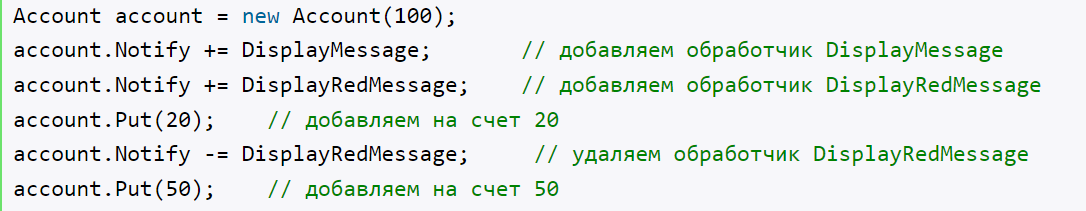




С событием может быть связан один или несколько обработчиков. Обработчики событий - это именно то, что выполняется при вызове событий. Нередко в качестве обработчиков событий применяются методы. Каждый обработчик событий по списку параметров и возвращаемому типу должен соответствовать делегату, который представляет событие. Для добавления обработчика события применяется операция **+=**:



Для одного события можно установить несколько обработчиков и потом в любой момент времени их удалить. Для удаления обработчиков применяется операция **-=**.

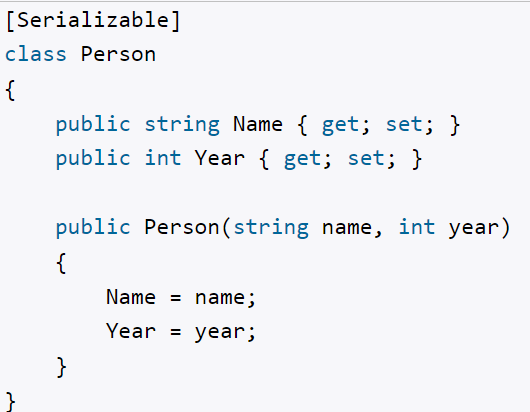


В качестве обработчиков могут использоваться не только обычные методы, но также делегаты, анонимные методы и лямбда-выражения.

18. Сериализация в языке С#. Стандартная и пользовательская сериализация.

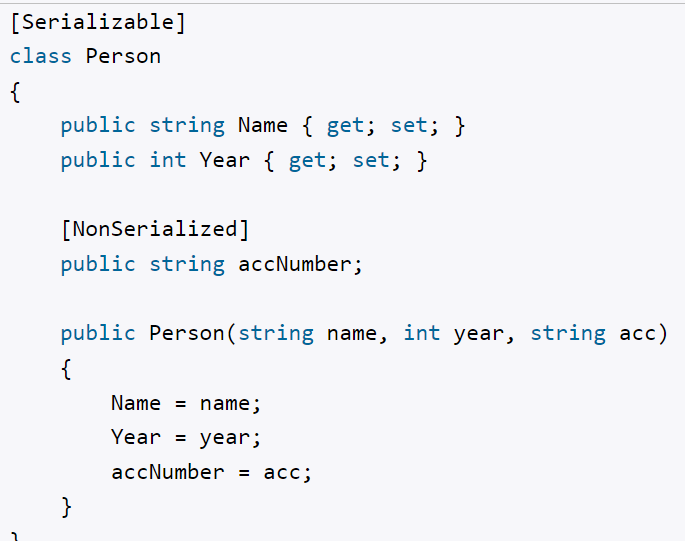
**Сериализация** представляет процесс преобразования какого-либо объекта в поток байтов. После преобразования мы можем этот поток байтов или записать на диск или сохранить его временно в памяти. А при необходимости можно выполнить обратный процесс - **десериализацию**, то есть получить из потока байтов ранее сохраненный объект.

Чтобы объект определенного класса можно было сериализовать, надо этот класс пометить атрибутом **Serializable**



При отстутствии данного атрибута объект Person не сможет быть сериализован, и при попытке сериализации будет выброшено исключение SerializationException.

Сериализация применяется к свойствам и полям класса. Если мы не хотим, чтобы какое-то поле класса сериализовалось, то мы его помечаем атрибутом **NonSerialized**:



При наследовании подобного класса, следует учитывать, что атрибут Serializable автоматически не наследуется. И если мы хотим, чтобы производный класс также мог бы быть сериализован, то опять же мы применяем к нему атрибут

Хотя сериализация представляет собой преобразование объекта в некоторый набор байтов, но в действительности только бинарным форматом она не ограничивается. Итак, в .NET можно использовать следующие форматы:

* бинарный
* SOAP
* xml
* JSON

Для каждого формата предусмотрен свой класс: для сериализации в бинарный формат - класс **BinaryFormatter**, для формата SOAP - класс **SoapFormatter**, для xml - **XmlSerializer**, для json - **DataContractJsonSerializer**.

Используя пользовательскую сериализацию, вы можете точно указать, какие объекты и как будут сериализованы. Класс должен иметь отметку SerializableAttribute и реализовывать интерфейс ISerializable. Если вы хотите настраивать и десериализацию объекта, следует использовать пользовательский конструктор.

Базовая сериализация использует платформу .NET для автоматической сериализации объекта. Единственное требование заключается в необходимости применить атрибут SerializableAttribute к классу. Атрибут NonSerializedAttribute также можно использовать для исключения из сериализации определенных полей.

При использовании базовой сериализации управление версиями объектов может привести к ошибкам. Если проблемы с управлением версиями важны, рекомендуется использовать пользовательскую сериализацию. Базовая сериализация является самым простым способом сериализации, но не дает почти никакого контроля над процессом.

**19. Класс Array в языке С#. Основные свойства и методы класса.**

Все массивы в C# построены на основе класса Array из пространства имен System.

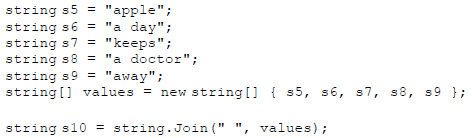
Основные свойства и методы:

* Свойство Length возвращает длину массива
* Свойство Rank возвращает размерность массива
* int BinarySearch (Array array, object? value) выполняет бинарный поиск в отсортированном массиве и возвращает индекс найденного элемента
* void Clear (Array array) очищает массив, устанавливая для всех его элементов значение по умолчанию
* void Copy (Array sourceArray, int sourceIndex, Array destinationArray, int destinationIndex, int length) копирует из массива sourceArray начиная с индекс sourceIndex length элементов в массив destinationArray начиная с индекса destinationIndex
* bool Exists<T> (T[] array, Predicate<T> match) проверяет, содержит ли массив array элементы, которые удовлеворяют условию делегата match
* void Fill<T> (T[] array, T value) заполняет массив array значением value
* T? Find<T> (T[] array, Predicate<T> match) находит первый элемент, который удовлеворяет определенному условию из делегата match. Если элемент не найден, то возвращается null
* T? FindLast<T> (T[] array, Predicate<T> match) находит последний элемент, который удовлеворяет определенному условию из делегата match. Если элемент не найден, то возвращается null
* int FindIndex<T> (T[] array, Predicate<T> match) возвращает индекс первого вхождения элемента, который удовлеворяет определенному условию делегата match
* int FindLastIndex<T> (T[] array, Predicate<T> match) возвращает индекс последнего вхождения элемента, который удовлеворяет определенному условию
* T[] FindAll<T> (T[] array, Predicate<T> match) возвращает все элементы в виде массива, которые удовлеворяет определенному условию из делегата match
* int IndexOf (Array array, object? value) возвращает индекс первого вхождения элемента в массив
* int LastIndexOf (Array array, object? value) возвращает индекс последнего вхождения элемента в массив
* void Resize<T> (ref T[]? array, int newSize) изменяет размер одномерного массива
* void Reverse (Array array) располагает элементы массива в обратном порядке
* void Sort (Array array) сортирует элементы одномерного массива

**20. Работа со строками в языке С#.**

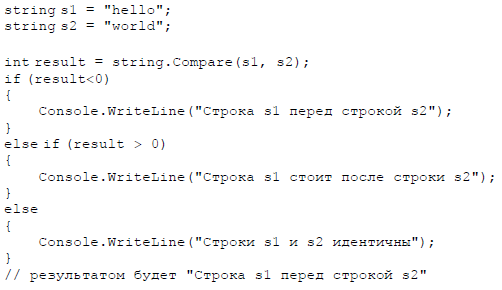
Этот тип определяет и поддерживает символьные строки. В C# строки являются объектами. Тип string относится к числу ссылочных.

**Конкатенация строк или объединение** может производиться как с помощью операции +, так и с помощью метода Concat: string s = string.Concat(имя\_строки,имя\_строки2 или же “Строка”)

Для объединения строк также может использоваться метод Join: 

**Сравнение строк**

Для сравнения строк применяется статический метод **Compare**:

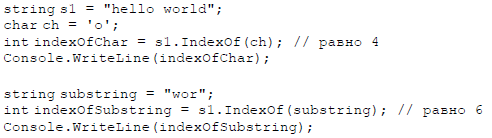


Данная версия метода Compare принимает две строки и возвращает число. Если первая строка по алфавиту стоит выше второй, то возвращается число меньше нуля. В противном случае возвращается число больше нуля. И третий случай - если строки равны, то возвращается число 0.

В данном случае так как символ h по алфавиту стоит выше символа w, то и первая строка будет стоять выше.

**Поиск в строке**

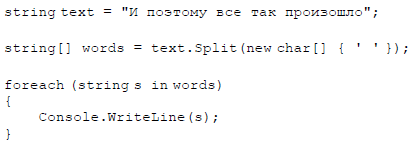
С помощью метода **IndexOf** мы можем определить индекс первого вхождения отдельного символа или подстроки в строке:



Подобным образом действует метод **LastIndexOf**, только находит индекс последнего вхождения символа или подстроки в строку.

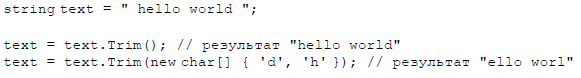
**Разделение строк**

С помощью функции Split мы можем разделить строку на массив подстрок. В качестве параметра функция Split принимает массив символов или строк, которые и будут служить разделителями. Например, подсчитаем количество слов в сроке, разделив ее по пробельным символам:

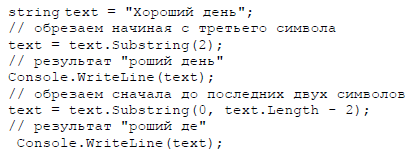


**Обрезка строки**

Для обрезки начальных или концевых символов используется функция **Trim**:



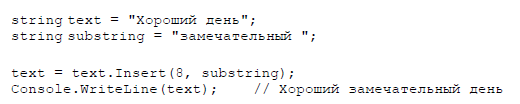
Обрезать определенную часть строки позволяет функция **Substring**:



В качестве параметра первая использованная версия применяет индекс, начиная с которого надо обрезать строку. Вторая версия применяет два параметра - индекс начала обрезки и длину вырезаемой части строки.

**Вставка**

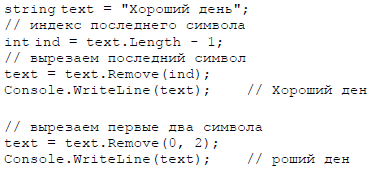
Для вставки одной строки в другую применяется функция **Insert**:



Первым параметром в функции Insert является индекс, по которому надо вставлять подстроку, а второй параметр - собственно подстрока.

**Удаление строк**

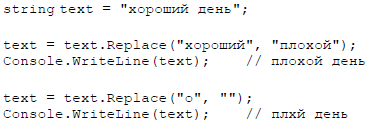
Удалить часть строки помогает метод **Remove**:



Первая версия метода Remove принимает индекс в строке, начиная с которого надо удалить все символы. Вторая версия принимает еще один параметр - сколько символов надо удалить.

**Замена**

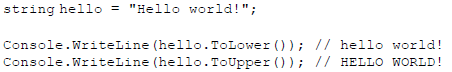
Чтобы заменить один символ или подстроку на другую, применяется метод **Replace**:



Во втором случае применения функции Replace строка из одного символа "о" заменяется на пустую строку, то есть фактически удаляется из текста. Подобным способом легко удалять какой-то определенный текст в строках.

**Смена регистра**

Для приведения строки к верхнему и нижнему регистру используются соответственно функции **ToUpper()** и **ToLower()**:



**21. Язык UML. Основные понятия.**

UML, или Unified Modeling Language, — унифицированный язык моделирования. Это графический язык, который с помощью диаграмм и схем описывает разнообразные процессы и структуры.

Важная особенность UML — этот язык поддерживает объектно-ориентированный подход, где все сущности представлены как объекты с определенными свойствами и методами. В диаграммах UML легко изобразить объекты, связи между ними, наследование и возможности передачи данных от одного объекта к другому.

Схема UML — концептуальная: это значит, что она оперирует концепциями и связями между ними. Сама диаграмма состоит из фигур, значков, надписей, линий и контуров.

3 вида uml-диаграмм:

* Объектные, или структурные — диаграммы, которые описывают статическую структуру системы. Это, например, диаграммы классов в программе. В них показываются сами объекты и классы, связи между ними, зависимости и атрибуты.
* Поведенческие — схемы для описания поведения в динамике. Их еще называют динамическими. Это, например, диаграммы, которые описывают процессы.
* Функциональные — диаграммы, которые описывают особенности функционирования, показывают, как устроена работа того или иного объекта. Они демонстрируют пользователю функциональность так, чтобы это было понятно. Обычно это именно физические особенности реализации функций, поэтому еще их называют диаграммами реализаций. Разные функциональные диаграммы можно отнести к объектным или поведенческим.

Модули для IDE и библиотеки для популярных языков программирования позволяют создавать UML с помощью программного кода. Тут тоже есть две версии использования: сгенерировать диаграмму на основе имеющегося кода проекта или запрограммировать ее вручную. Можно описывать на языке программирования элементы и связи между ними, а потом «рисовать» визуализацию с помощью других инструментов. Такие утилиты и библиотеки существуют, например, для Java, JavaScript, PHP и других распространенных языков.

**22. Шаблоны проектирования. Примеры.**

Шаблоны проектирования = Паттерны.

Паттерн представляет определенный способ построения программного кода для решения часто встречающихся проблем проектирования. В данном случае предполагается, что есть некоторый набор общих формализованных проблем, которые довольно часто встречаются, и паттерны предоставляют ряд принципов для решения этих проблем.



**Пример:**

Порождающий паттерн

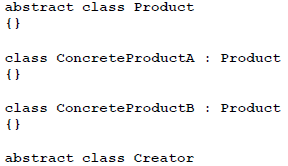
Фабричный метод (Factory Method)

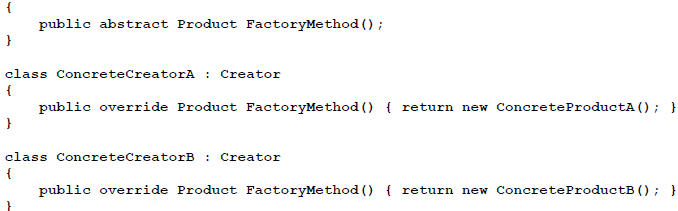
Фабричный метод (Factory Method) - это паттерн, который определяет интерфейс для создания объектов некоторого класса, но непосредственное решение о том, объект какого класса создавать происходит в подклассах. То есть паттерн предполагает, что базовый класс делегирует создание объектов классам-наследникам.

Когда надо применять паттерн:

* Когда заранее неизвестно, объекты каких типов необходимо создавать
* Когда система должна быть независимой от процесса создания новых объектов и расширяемой: в нее можно легко вводить новые классы, объекты которых система должна создавать.
* Когда создание новых объектов необходимо делегировать из базового класса классам наследникам

Формальное определение паттерна на языке C# может выглядеть следующим образом:





* Абстрактный класс Product определяет интерфейс класса, объекты которого надо создавать.
* Конкретные классы ConcreteProductA и ConcreteProductB представляют реализацию класса Product. Таких классов может быть множество
* Абстрактный класс Creator определяет абстрактный фабричный метод FactoryMethod(), который возвращает объект Product.
* Конкретные классы ConcreteCreatorA и ConcreteCreatorB - наследники класса Creator, определяющие свою реализацию метода FactoryMethod(). Причем метод FactoryMethod() каждого отдельного класса-создателя возвращает определенный конкретный тип продукта. Для каждого конкретного класса продукта определяется свой конкретный класс создателя.

Таким образом, класс Creator делегирует создание объекта Product своим наследникам. А классы ConcreteCreatorA и ConcreteCreatorB могут самостоятельно выбирать какой конкретный тип продукта им создавать.