Инкапсуляция - реализация механизма скрытия внутренней реализации объекта. Упрощает программирование и запрещает несанкционированный доступ к данным, которые должны быть закрыты. Программирование методом черного ящика: хорошо инкапсулированный класс должен скрывать свои необработанные данные и то, как он ими манипулирует.Способы реализации:

1. пары традиционных методов доступа и изменения
2. именованные свойства

Здесь менять значение переменной может только текущий класс или производный класс и не может быть изменено из экземпляра класса. Можно также создавать свойства только для чтения или для записи опускают соответственно блок set или get. Для изменения значения свойств поля только для чтения нужно использовать аргументы конструктора.
Статические свойства, как и все статические члены, доступны на уровне класса, а не на уровне экземпляра и используется так же, как и статические методы. Если нужно гарантировать, что статическое свойство будет всегда установлено в одно и то же значение используется статический конструктор (не имеет модификаторов доступа).Статические свойства должны оперировать статическими данными.
Наследование - создание новых классов на основе уже существующих. Расширяет поведение базового (родительского) класса, разрешая подклассу наследовать основную функциональность — отношение классического наследования. Существует еще модель включение/делегирования, модель отношения «имеет»: созданный класс может определять переменную-член другого класса и предоставлять часть своей функциональности или всю функциональность внешнему миру.
С помощью ключевого слова base можно вызывать пользовательский конструктор базового класса:

```
public class Manager: Employee
{
public Manager(string name, int age, int id, ulong numOfOpts)
:base(name, age, id)//вызов пользовательского конструктора Employee
{
nimberOfOprions = numOfOpts; //устанавливет свойство текущего класса
}
}
```

модификатор protected — дочерние классы могут напрямую обращаться к этой информации, чего не могут делать пользователи объекта.
модификатор sealed — гарантирует, что класс не может выступать в качестве базового.
Полиморфизм - возможность языка использовать связанные объекты одиноковым образом. Базовый класс может определить набор членов (полиморфный интерфейс) для всех своих наследников. Полиморфный интерфейс класса создается с любым количеством виртуальных или абстрактных членов. Виртуальный член может быть изменен (перекрыт) производным классом, а абстрактный должен быть перекрыт производным классом. Если в базовом классе определяется метод, который может быть перекрыт подклассом, этот метод должен быть виртуальным.

```
public class Employee
{
public virtual GiveBonus(float amount)
{currPay += amount;}
}
```

Если подклассу нужно перекрыть этот метод это делается с помощью ключевого слова override.

```
public override GiveBous(float amount)
{
```

```
//некоторая новая логика
//также может быть использована старая логика с помощью ключевого слова base
base.GiveBonus(amount);
}
```

Если на каком-то этапе нужно гарантировать, что виртуальный метод больше не будет перекрыт, то можно поставить модификатор sealed:

```
public override sealed void GiveBonus(float amount)
{…}
```

Чтобы предотвратить возможность создания базового объекта, не несущего особой смысловой нагрузки, помимо объединения некоторых общих свойств и функционала нужно использовать ключевое слово abstract:

```
Abstract public class Employee
{….}
```

В случае попытки создание объекта данного класса получим ошибку времени компиляции. Абстрактный базовый класс может содержать любое количество абстрактных членов. Абстрактные методы могут использоваться для определения методов, которые по умолчанию не предоставляют реализацию. Используются для определения полиморфных черт всех наследников, оставляя на их усмотрение реализацию. Абстрактные методы могу определяться только в абстрактных классах.
Очень полезным также может оказаться перекрытие методов с помощью ключевого слова new:

```
Public class ThteeDCircle: Circle
{
New protected string point;
//скрыть любые реализации Draw(), созданные ранее
public new Draw()
{Console.WriteLine("new draw 3d");}
}
```

Открытый интерфейс — множество членов, которые непосредственно доступны из объектной переменной с помощью оператора точка. С точки зрения класса — это любой член, объявленный в классе с использованием ключевого слова public. Может содержать поля данных, конструкторы, методы, свойства, костанты и поля только для чтеня.
Объект. Теоретические сведения ключевое слово new отвечает за вычисление правильного количества байт для указанного объекта и выделение досточного объема памяти из управляемой кучи.
Объектные переменные представляют собой ссылки на объекты в памяти, а не сами объекты.
Принцип разделения ответственности: один тип работает как объект приложения (определяет метод Main()) и множество других типов, составляющих все приложение.
При определении ЛОКАЛЬНЫХ переменных присвоение начальных значений перед использованием ОБЯЗАТЕЛЬНО, т.к. они не получают значений по умолчанию.
Static методы могут быть вызваны на уравне класса, а не на уравне экземпляра. Могут использовать только статические данные.
Адреса объектов определяются во время выполнения, а значение const полей устанавливается на этапе компиляции.

Интерфейсы как элемент ООП.

Для объявления интерфейса используется ключевое слово `interface`. Интерфейс содержит только заголовки методов, свойств и событий. Для свойства указываются только ключевые слова `get` и (или) `set`. При объявлении элементов интерфейса не могут использоваться следующие модификаторы:

`abstract`, `public`, `protected`, `internal`, `private`, `virtual`, `override`, `static`. Считается, что все элементы интерфейса имеют `public`-уровень доступа:
Если класс является производным от некоторого базового класса, то имя базового класса указывается перед именем реализуемого интерфейса.

Элементы интерфейса допускают явную и неявную реализацию. При *неявной реализации* тип должен содержать открытые экземплярные элементы, имена и сигнатура которых соответствуют элементам интерфейса. При *явной реализации* элемент типа называется по форме *<имя интерфейса>.<имя элемента>*, а указание любых модификаторов для элемента при этом запрещается. Все неявно реализуемые элементы интерфейса по умолчанию помечаются в классе как `sealed`. А значит, наследование классов не ведёт к прямому наследованию реализаций:
Чтобы осуществить наследование реализаций, требуется при реализации использовать модификаторы `virtual` и `override`[1].
Подобно классам, интерфейсы могут наследоваться от других интерфейсов. При этом наследование интерфейсов может быть множественным. Один класс может реализовывать несколько интерфейсов - имена интерфейсов перечисляются после имени класса через запятую.
Интерфейсы сходны с абстрактными классам, что иногда порождает проблему выбора «интерфейс или абстрактный класс?». Табл. 3 содержит сравнение возможностей этих пользовательских типов, для того, чтобы помочь сделать правильный выбор между ними.

Таблица 3

Сравнение абстрактных классов и интерфейсов

Абстрактные классы	**Интерфейсы**
Не могут быть созданы напрямую, но могут содержать конструктор, который вызывается в классе-наследнике	Не могут содержать конструктор
Абстрактный класс может быть так дополнен элементами, что это не повлияет на его классы-наследники	Если в интерфейс помещаются дополнительные элементы, все классы, которые его реализуют, должны быть дополнены
Может хранить данные в полях	Не может хранить данных
Виртуальные элементы могут содержать базовую реализацию. Допустимы невиртуальные элементы	Все элементы являются виртуальными и не включают реализацию
Класс может наследоваться от единственного абстрактного класса	Класс может реализовывать несколько интерфейсов
Класс-наследник может переопределить только некоторые элементы абстрактного класса	Класс, который реализует интерфейс, должен реализовать все элементы интерфейса
Наследование поддерживается только для классов	Интерфейс может быть реализован структурой

[1] При явной реализации использование модификаторов невозможно.