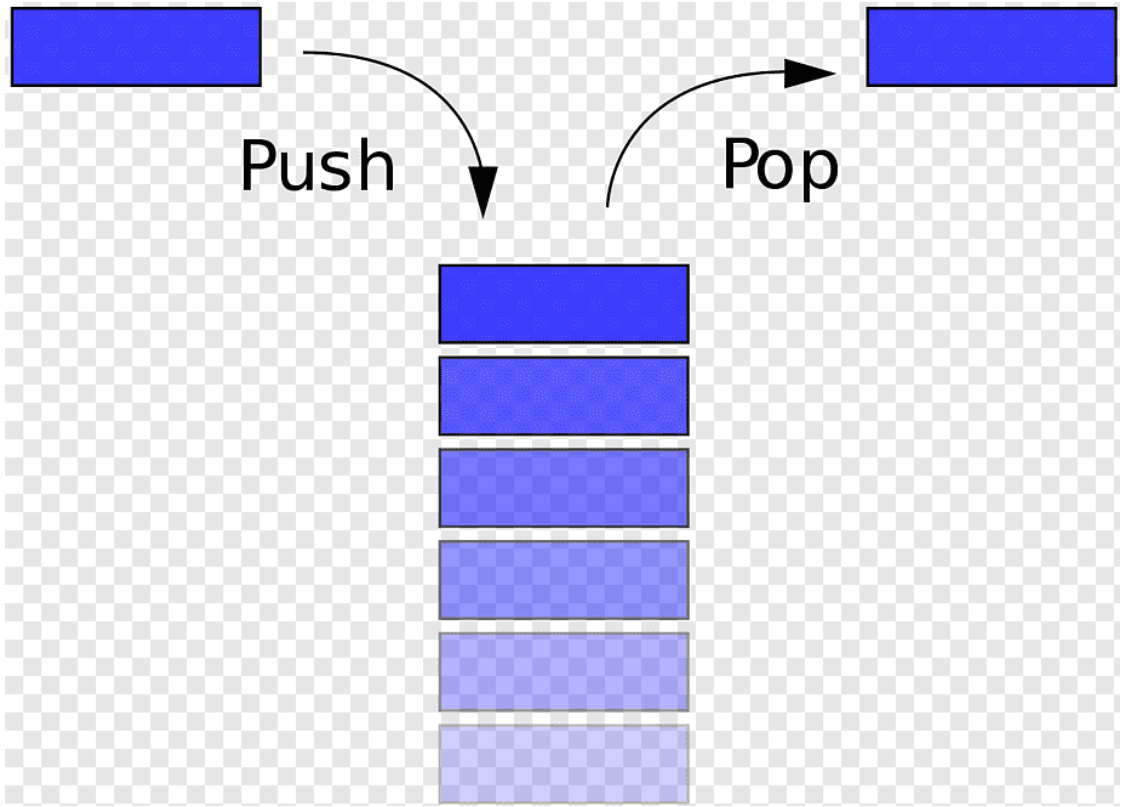
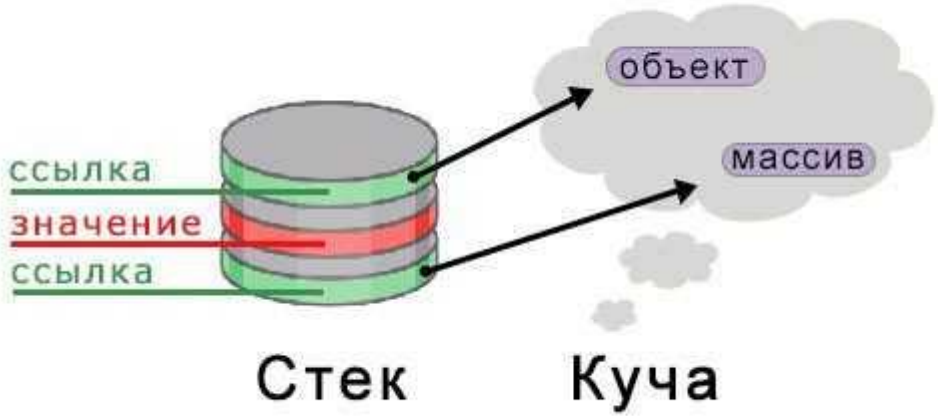
«Сборка мусора, управление памятью и указатели»

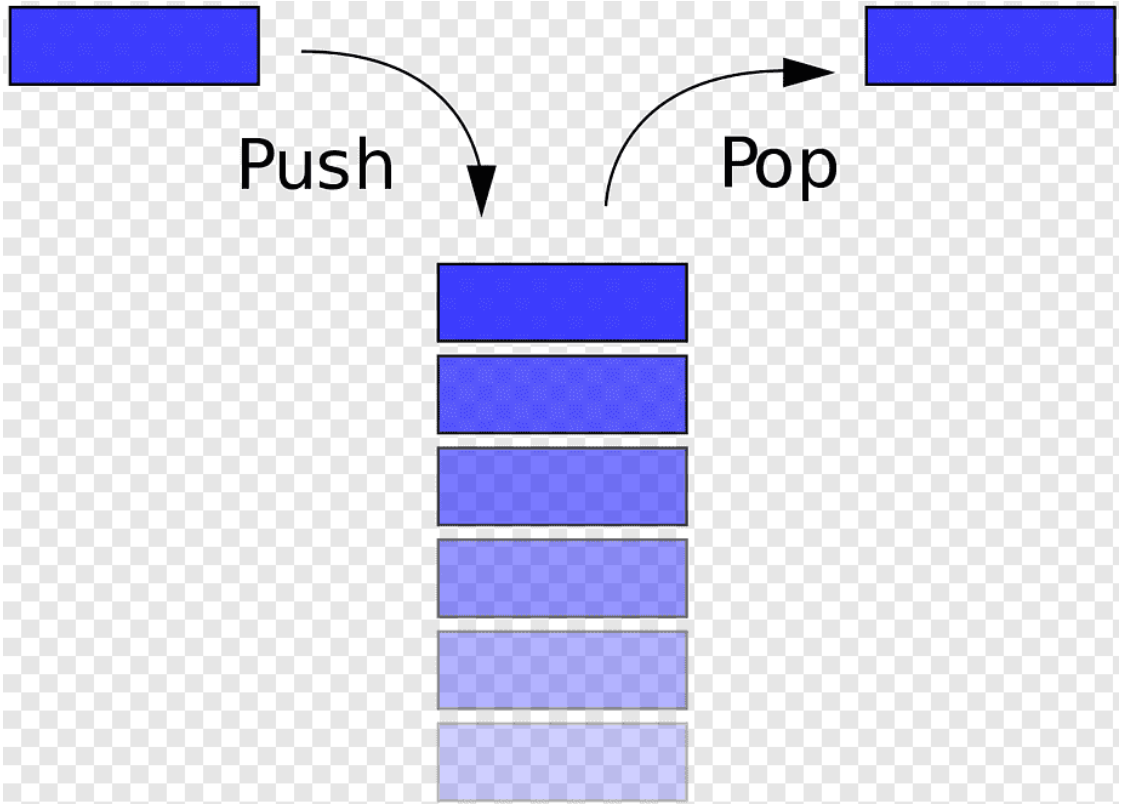
СБОРЩИК МУСОРА В C#

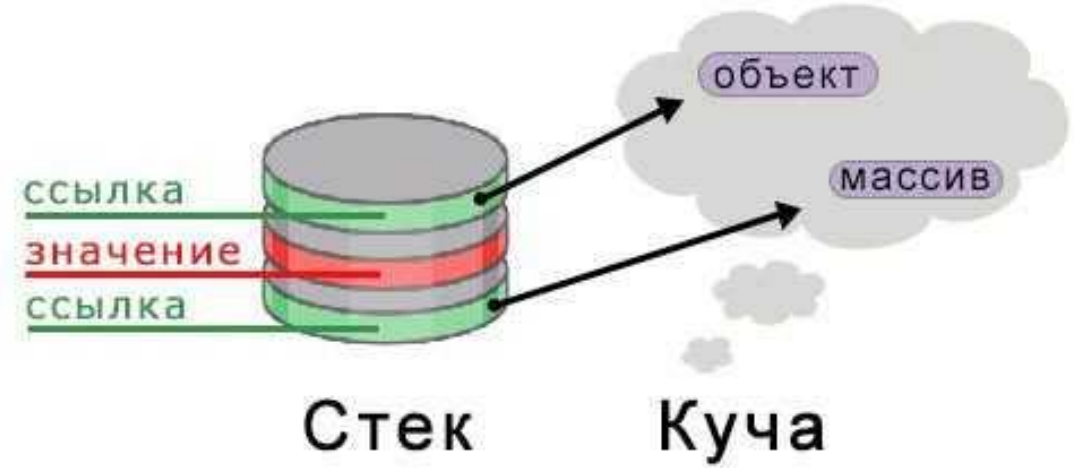
Стек /тэ/ — **абстрактный тип данных**, представляющий собой список элементов, организованных по принципу LIFO (last in — first out, «последним пришёл — первым вышел»).



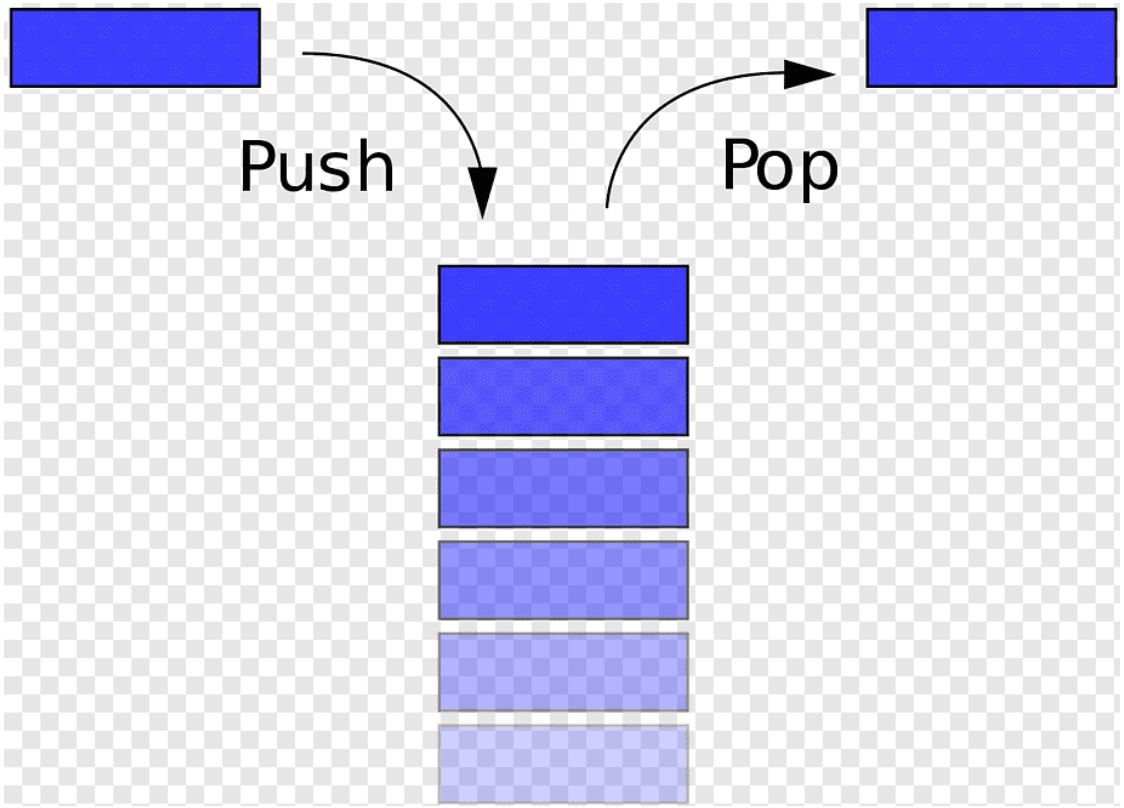


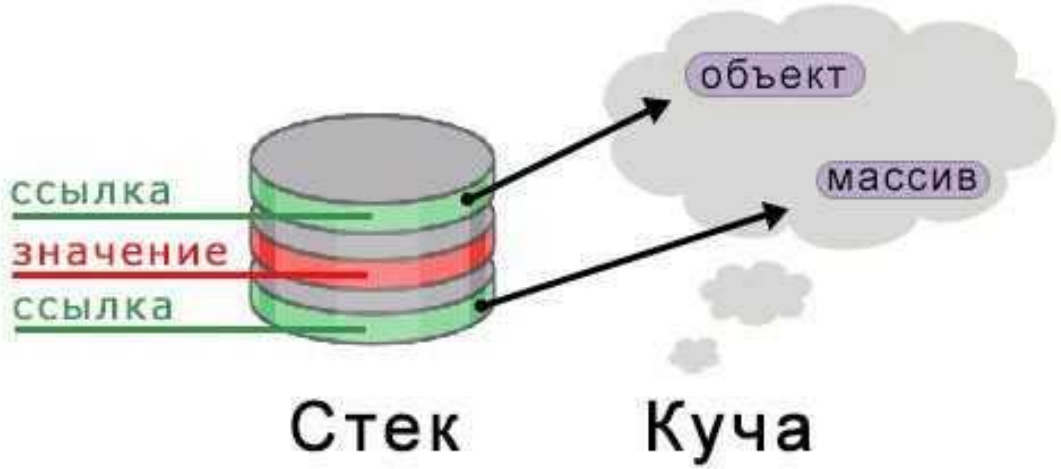
СБОРЩИК МУСОРА В C#
[КУЧА И CLR]





СБОРЩИК МУСОРА В C# [КУЧА И Common Language Runtime (CLR)]





СБОРЩИК МУСОРА В C#

При использовании ссылочных типов (объекты классов), для них будет отводиться место в **стеке*,** только там будет храниться не значение, а адрес на участок памяти, в котором и будут находиться сами значения данного объекта.

Сборщик мусора (garbage collector)

СБОРЩИК МУСОРА В C#

```
Test();

void Test()
{
    Person tom = new Person("Tom");
    Console.WriteLine(tom.Name);
}
record class Person(string Name);
```

СБОРЩИК МУСОРА В C#

```
Test();

void Test()
{
    Person tom = new Person("Tom");
    Console.WriteLine(tom.Name);
}
record class Person(string Name);
```

СБОРЩИК МУСОРА В C#
[КУЧА]

Как правило, объекты в куче располагаются неупорядочено, между ними могут иметься пустоты.
Куча сильно фрагментирована

СБОРЩИК МУСОРА В C#
[КУЧА]

Для крупных объектов существует своя куча - **Large Object Heap**. В эту кучу помещаются объекты, размер которых больше **85 000 байт.**

СБОРЩИК МУСОРА В C#
[КУЧА]

Чтобы снизить издержки от работы сборщика мусора, все объекты в куче разделяются **по поколениям**.
Всего существует три поколения объектов:
0, 1 и 2-е.

СБОРЩИК МУСОРА В C#
[КУЧА]

Чтобы снизить издержки от работы сборщика мусора, все объекты в куче разделяются **по поколениям**.
Всего существует три поколения объектов:
0, 1 и 2-е.

СБОРЩИК МУСОРА В C#
[КУЧА]

Чтобы снизить издержки от работы сборщика мусора, все объекты в куче разделяются **по поколениям**.
Всего существует три поколения объектов:
0, 1 и 2-е.

Класс System.GC

Функционал сборщика мусора в библиотеке классов .NET представляет класс **System.GC**.

Класс System.GC

Метод **AddMemoryPressure** информирует среду **CLR** о выделении большого объема памяти, которую надо учесть при планировании сборки мусора.

Класс System.GC

Метод **Collect** приводит в действие механизм сборки мусора.

Класс System.GC

Метод **GetGeneration(Object)** позволяет определить номер поколения, к которому относится переданный в качестве параметра объект

Класс System.GC

Метод **GetTotalMemory** возвращает объем памяти в байтах, которое занято в управляемой куче

Класс System.GC

Метод **WaitForPendingFinalizers** приостанавливает работу текущего потока до освобождения всех объектов, для которых производится сборка мусора

Класс System.GC

С помощью перегруженных версий **метода GC.Collect** можно выполнить более точную настройку сборки мусора.

Принимает в качестве параметра число **- номер поколения**, вплоть до которого надо выполнить очистку.

Например, **GC.Collect(0) -** удаляются только объекты **поколения 0**.

Класс System.GC

Параметр - перечисление **GCCollectionMode**. Это перечисление может принимать три значения:

Default: значение по умолчанию для данного перечисления (Forced)

Forced: вызывает немедленное выполнение сборки мусора

Optimized: позволяет сборщику мусора определить, является ли текущий текущий момент оптимальным для сборки мусора

ФИНАЛИЗИРУЕМЫЕ ОБЪЕКТЫ

Большинство объектов, используемых в программах на C#, относятся к **управляемым или managed-коду**.

Однако вместе с тем встречаются также и такие объекты, которые задействуют неуправляемые объекты (подключения к файлам, базам данных, сетевые подключения и т.д.).

ФИНАЛИЗИРУЕМЫЕ ОБЪЕКТЫ

Освобождение неуправляемых ресурсов подразумевает реализацию одного из двух механизмов:

1. Создание деструктора
2. Реализация классом интерфейса **System.IDisposable**

СОЗДАНИЕ ДЕСТРУКТОРОВ

Метод деструктора носит имя класса (как и конструктор), перед которым стоит знак тильды (~).

Деструкторы можно определить **только в классах**. Деструктор в отличие от конструктора **не может иметь модификаторов доступа и параметры**. При этом каждый **класс** может иметь только **один деструктор**.

СОЗДАНИЕ ДЕСТРУКТОРОВ

```
class Person
{
    public string Name { get;}
    public Person(string name) => Name = name;

    ~Person()
    {
        Console.WriteLine($"{Name} has deleted");
    }
}
```

СОЗДАНИЕ ДЕСТРУКТОРОВ

НА ДЕЛЕ ПРИ ОЧИСТКЕ СБОРЩИК МУСОРА ВЫЗЫВАЕТ НЕ ДЕСТРУКТОР, А МЕТОД **FINALIZE**

```
protected override void Finalize()
{
    try
    {
        // здесь идут инструкции деструктора
    }
    finally
    {
        base.Finalize();
    }
}
```

ИНТЕРФЕЙС IDISPOSABLE

Интерфейс **IDisposable** объявляет один единственный метод **Dispose**, в котором при реализации интерфейса в классе должно происходить освобождение неуправляемых ресурсов.

```
try
{
    tom = new Person("Tom");
}
finally
{
    tom?.Dispose();
}
```

```
Person tom = new Person("Tom");
tom.Dispose();
```

ОБЩИЕ РЕКОМЕНДАЦИИ ПО ИСПОЛЬЗОВАНИЮ FINALIZE И DISPOSE

- **Деструктор** следует **реализовывать** только у тех объектов, которым он действительно **необходим**, так как метод Finalize оказывает сильное **влияние** на **производительность**

ОБЩИЕ РЕКОМЕНДАЦИИ ПО ИСПОЛЬЗОВАНИЮ FINALIZE И DISPOSE

- После вызова метода Dispose необходимо блокировать у объекта вызов метода Finalize с помощью GC.SuppressFinalize

ОБЩИЕ РЕКОМЕНДАЦИИ ПО ИСПОЛЬЗОВАНИЮ FINALIZE И DISPOSE

- При создании производных классов от базовых, которые реализуют интерфейс IDisposable

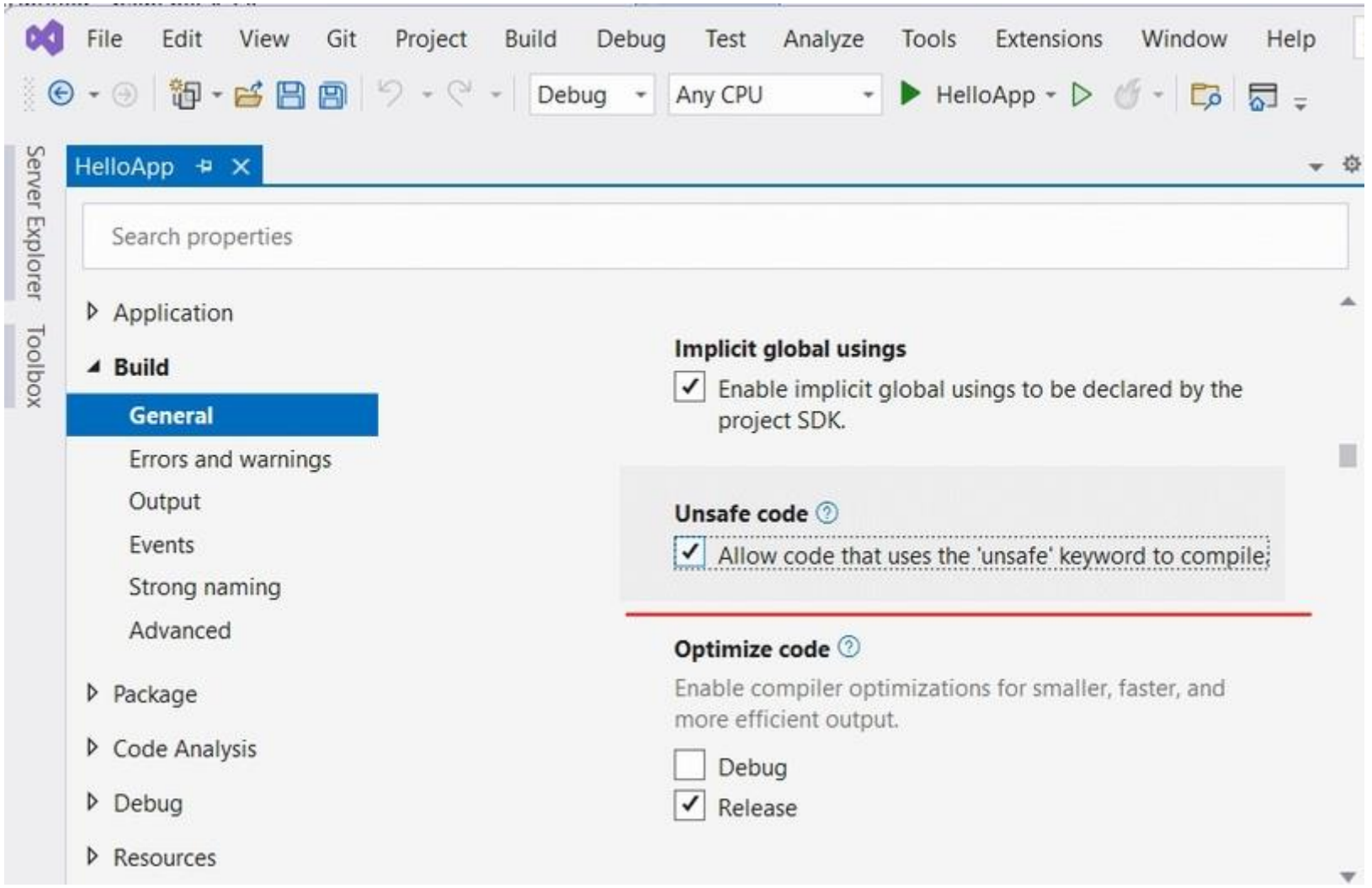
КОНСТРУКЦИЯ USING

```
using (Person tom = new Person("Tom"))
{
}
```

УКАЗАТЕЛИ

Код, применяющий указатели, еще называют небезопасным (**unsafe**) кодом.

УКАЗАТЕЛИ


File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help
Debug
Any CPU
HelloApp
Server Explorer
Toolbox
HelloApp
Search properties
Application
Build
General
Errors and warnings
Output
Events
Strong naming
Advanced
Package
Code Analysis
Debug
Resources
Implicit global usings
Enable implicit global usings to be declared by the project SDK.
Unsafe code
Allow code that uses the 'unsafe' keyword to compile
Optimize code
Enable compiler optimizations for smaller, faster, and more efficient output.
Debug
Release

КЛЮЧЕВОЕ СЛОВО UNSAFE

Блок кода или метод, в котором используются указатели, помечается ключевым словом **unsafe**:

```
// блок кода, использующий указатели
unsafe
{

}
```

Метод, использующий указатели:

```
unsafe void Test()
{

}
```

ОПЕРАЦИИ * И &

Ключевой при работе с указателями является **операция ***, которую еще называют **операцией разыменовывания**. Операция разыменовывания **позволяет получить** или установить **значение по адресу**, на который указывает указатель. Для получения **адреса переменной** применяется **операция &**

ОПЕРАЦИИ * И &

Ключевой при работе с указателями является **операция ***, которую еще называют **операцией разыменовывания**. Операция разыменовывания **позволяет получить** или установить **значение по адресу**, на который указывает указатель. Для получения **адреса переменной** применяется **операция &**

ОПЕРАЦИИ * И &

```
unsafe
{
    int* x; // определение указателя
    int y = 10; // определяем переменную

    x = &y; // указатель x теперь указывает на адрес переменной y
    Console.WriteLine(*x); // 10

    y = y + 20;      // меняем значение
    Console.WriteLine(*x);// 30

    *x = 50;
    Console.WriteLine(y); // переменная y=50
}
```

ПОЛУЧЕНИЕ АДРЕСА

```
int* x; // определение указателя
int y = 10; // определяем переменную

x = &y; // указатель x теперь указывает на адрес переменной y

// получим адрес переменной y
ulong addr = (ulong)x;
Console.WriteLine($"Адрес переменной y: {addr}");
```

УКАЗАТЕЛИ НА СТРУКТУРЫ, ЧЛЕНЫ КЛАССОВ И МАССИВЫ

Кроме указателей на простые типы можно использовать **указатели на структуры**. А для доступа к полям структуры, на которую указывает указатель, используется операция **->**

```
    }
    public override string ToString() => $"X: {X}  Y: {Y}";
}
```

УКАЗАТЕЛИ НА МАССИВЫ И STACKALLOC

С помощью ключевого слова **stackalloc** можно выделить память под массив в стеке. Смысл выделения памяти в стеке в повышении быстродействия кода.

«Основы паттернов проектирования. Часть 1»

ВВЕДЕНИЕ В ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

Паттерн представляет определенный способ построения программного кода для решения часто встречающихся проблем проектирования.

Design Patterns: Elements of Reusable Object-Oriented Software

Что же дает нам применение паттернов?

При написании программ мы можем формализовать проблему в виде классов и объектов и связей между ними. И применить один из существующих паттернов для ее решения.

Что же дает нам применение паттернов?

Причем паттерны, как правило, не зависят от языка программирования. Их принципы применения будут аналогичны и в C#, и в Java, и в других языках.

Паттерны

Порождающие паттерны — это паттерны, которые абстрагируют процесс **инстанцирования** или, иными словами, процесс порождения классов и объектов. Среди них выделяются следующие.

- Абстрактная фабрика (Abstract Factory)
- **Строитель (Builder)**
- Фабричный метод (Factory Method)
- **Прототип (Prototype)**
- Одиночка (Singleton)

Паттерны

Структурные паттерны - рассматривает, как классы и объекты образуют более крупные структуры - более сложные по характеру классы и объекты. К таким шаблонам относятся:

- Адаптер (Adapter)
- **Мост (Bridge)**
- Компоновщик (Composite)
- Декоратор (Decorator)
- **Фасад (Facade)**
- Приспособленец (Flyweight)
- **Заместитель (Proxy)**

Паттерны

Поведенческими - они определяют алгоритмы и взаимодействие между классами и объектами, то есть их поведение. Среди подобных шаблонов можно выделить следующие

Паттерны

- **Цепочка обязанностей (Chain of responsibility)**
- Команда (Command)
- **Интерпретатор (Interpreter)**
- Итератор (Iterator)
- **Посредник (Mediator)**
- Хранитель (Memento)
- **Наблюдатель (Observer)**
- Состояние (State)
- **Стратегия (Strategy)**
- Шаблонный метод (Template method)
- **Посетитель (Visitor)**

Паттерны

- **Паттерны классов** описывают отношения между классами посредством наследования.

Отношения между классами определяются на стадии компиляции.

- **Фабричный метод (Factory Method)**
- Интерпретатор (Interpreter)
- **Шаблонный метод (Template Method)**
- Адаптер (Adapter)

Паттерны

- **Паттерны объектов** описывают отношения между объектами. Эти отношения возникают на этапе выполнения, поэтому обладают большей гибкостью.

Как выбрать нужный паттерн?

- **Выделить все используемые сущности и связи между ними и абстрагировать их от конкретной ситуации**

- **Посмотреть, вписывается ли абстрактная форма решения задачи в определенный паттерн**

(Например, суть решаемой задачи может состоять в создании новых объектов. В этом случае, возможно, стоит посмотреть на ***порождающие паттерны***.)

Как выбрать нужный паттерн?

ВАЖНО

понимать смысл и назначение паттерна, явно представлять его абстрактную организацию и его возможные конкретные реализации.

Отношения между классами и объектами

Основные отношения:

наследование, реализация, ассоциация, композиция и агрегация

Отношения между классами и объектами

Наследование

Наследование является базовым принципом ООП и позволяет одному классу (наследнику) унаследовать функционал другого класса (родительского).

называют **генерализацией или обобщением**

```
class User
{
    public int Id { get; set; }
    public string Name { get; set; }
}

class Manager : User
{
    public string Company{ get; set; }
}
```

Отношения между классами и объектами

Наследование

С помощью **диаграмм UML** отношение между классами выражается в не закрашенной стрелочке от класса-наследника к классу-родителю

Отношения между классами и объектами

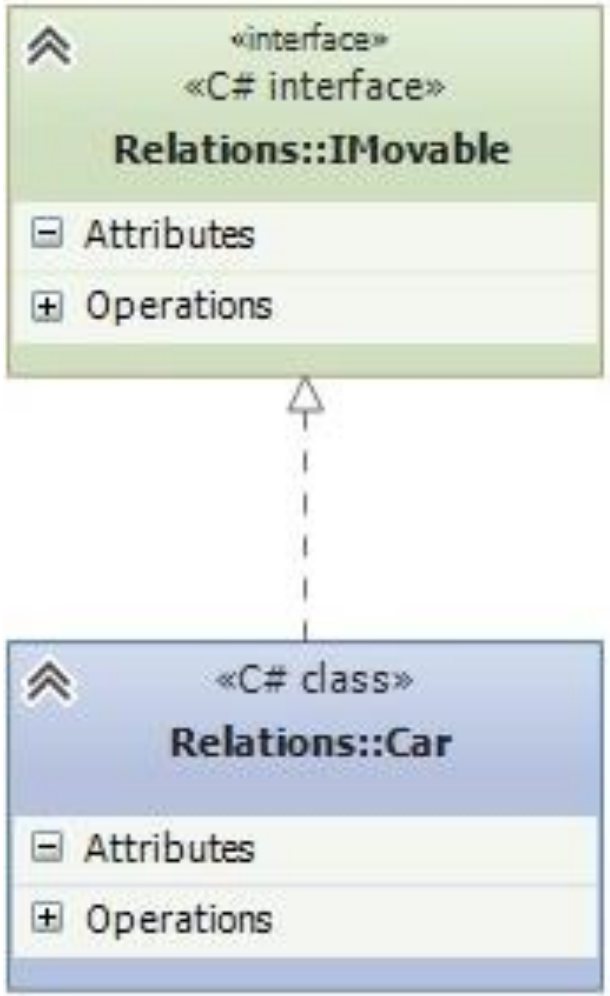
Реализация

Реализация предполагает определение интерфейса и его реализация в классах. Например, имеется **интерфейс IMovable** с **методом Move**, который реализуется в **классе Car**

```
public interface IMovable
{
    void Move();
}
public class Car : IMovable
{
    public void Move()
    {
        Console.WriteLine("Машина едет");
    }
}
```

Отношения между классами и объектами

Реализация



С помощью **диаграмм UML** отношение реализации также выражается в не закрашенной стрелочке от класса к интерфейсу, только линия теперь **пунктирная**

Отношения между классами и объектами

Ассоциация

Ассоциация - это отношение, при котором объекты одного типа неким образом связаны с объектами другого типа.
(*Например, объект одного типа содержит или использует объект другого типа. Например,* ***игрок*** *играет в определенной команде*)

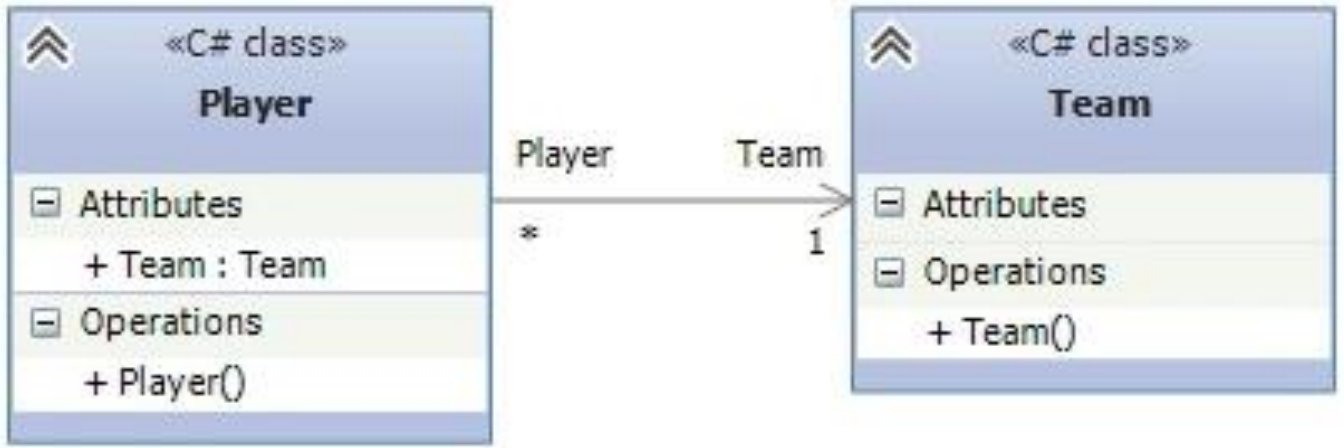
```
class Team
{

}
class Player
{
    public Team Team { get; set; }
}
```

Отношения между классами и объектами

Ассоциация

Класс Player связан отношением ассоциации с класом Team. На схемах UML ассоциация обозначается в виде обычно стрелки:



Отношения между классами и объектами

Композиция

Композиция определяет отношение **HAS A**, то есть отношение "**имеет**". (*Например, в класс автомобиля содержит объект класса электрического двигателя*):

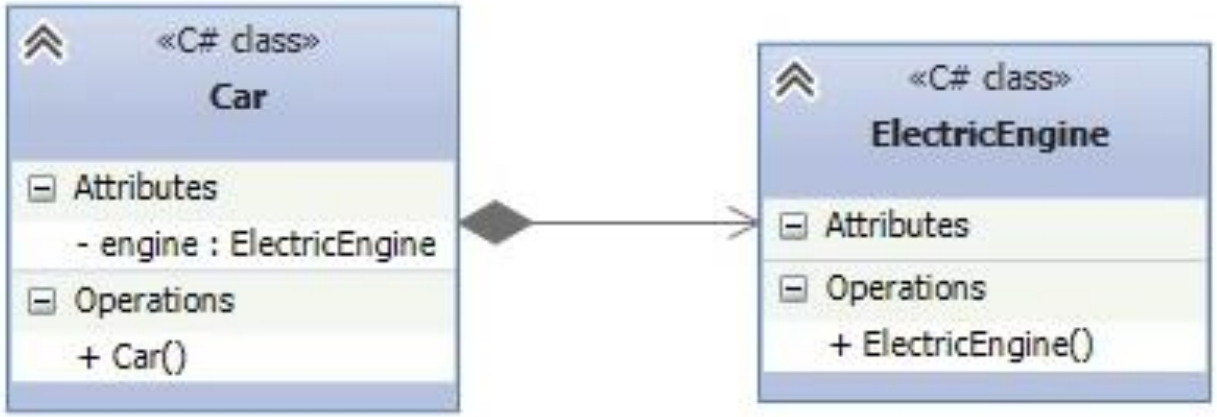
```
public class ElectricEngine
{ }

public class Car
{
    ElectricEngine engine;
    public Car()
    {
        engine = new ElectricEngine();
    }
}
```

Отношения между классами и объектами

Композиция

На диаграммах UML отношение композиции проявляется в обычной стрелке от главной сущности к зависимой, при этом со стороны главной сущности, которая содержит, объект второй сущности, располагается закрашенный ромбик:



Отношения между классами и объектами

Агрегация

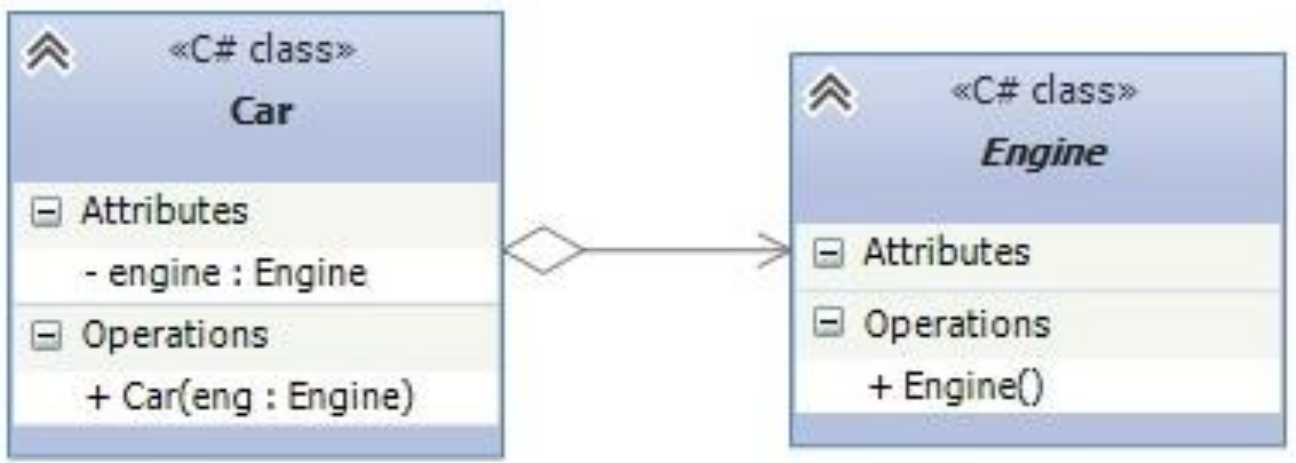
```
public abstract class Engine
{ }

public class Car
{
    Engine engine;
    public Car(Engine eng)
    {
        engine = eng;
    }
}
```

Отношения между классами и объектами

Агрегация

Отношение агрегации на диаграммах UML отображается также, как и отношение композиции, только теперь ромбик будет незакрашенным:



ИНТЕРФЕЙСЫ ИЛИ АБСТРАКТНЫЕ КЛАССЫ

Когда следует использовать абстрактные классы:

- Если надо определить общий функционал для родственных объектов
- Если мы проектируем довольно большую функциональную единицу, которая содержит много базового функционала
- Если нужно, чтобы все производные классы на всех уровнях наследования имели некоторую общую реализацию. При использовании абстрактных классов, если мы захотим изменить базовый функционал во всех наследниках, то достаточно поменять его в абстрактном базовом классе.
- Если же нам вдруг надо будет поменять название или параметры метода интерфейса, то придется вносить изменения и также во всех классы, которые данный интерфейс реализуют.

ИНТЕРФЕЙСЫ ИЛИ АБСТРАКТНЫЕ КЛАССЫ

```
public abstract class Vehicle
{
    public abstract void Move();
}

public class Car : Vehicle
{
    public override void Move()
    {
        Console.WriteLine("Машина едет");
    }
}

public class Bus : Vehicle
{
    public override void Move()
    {
        Console.WriteLine("Автобус едет");
    }
}

public class Tram : Vehicle
{
    public override void Move()
    {
        Console.WriteLine("Трамвай едет");
    }
}
```

ИНТЕРФЕЙСЫ ИЛИ АБСТРАКТНЫЕ КЛАССЫ

Когда следует использовать интерфейсы:

- Если нам надо определить функционал для группы разрозненных объектов, которые могут быть никак не связаны между собой.
- Если мы проектируем небольшой функциональный тип

ИНТЕРФЕЙСЫ ИЛИ АБСТРАКТНЫЕ КЛАССЫ

```
public interface IMovable
{
    void Move();
}

public abstract class Vehicle : IMovable
{
    public abstract void Move();
}

public class Car : Vehicle
{
    public override void Move() => Console.WriteLine("Машина едет");
}

public class Bus : Vehicle
{
    public override void Move() => Console.WriteLine("Автобус едет");
}

public class Hourse : IMovable
{
    public void Move() => Console.WriteLine("Лошадь скачет");
}

public class Aircraft : IMovable
{
    public void Move() => Console.WriteLine("Самолет летит");
}
```

ПОРОЖДАЮЩИЕ ПАТТЕРНЫ

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.

ПОРОЖДАЮЩИЕ ПАТТЕРНЫ

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

ПОРОЖДАЮЩИЕ ПАТТЕРНЫ

```
abstract class Product
{}

class ConcreteProductA : Product
{}

class ConcreteProductB : Product
{}

abstract class Creator
{
    public abstract Product FactoryMethod();
}

class ConcreteCreatorA : Creator
{
    public override Product FactoryMethod() { return new ConcreteProductA(); }
}

class ConcreteCreatorB : Creator
{
    public override Product FactoryMethod() { return new ConcreteProductB(); }
}
```

АБСТРАКТНАЯ ФАБРИКА

Паттерн "Абстрактная фабрика" (Abstract Factory) предоставляет интерфейс для создания семейств взаимосвязанных объектов с определенными интерфейсами без указания конкретных типов данных объектов.

АБСТРАКТНАЯ ФАБРИКА

Когда использовать абстрактную фабрику

- Когда система не должна зависеть от способа создания и компоновки новых объектов
- Когда создаваемые объекты должны использоваться вместе и являются взаимосвязанными

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.

ОДИНОЧКА

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.

ПРОТОТИП (PROTOTYPE)

Паттерн Прототип (Prototype) позволяет создавать объекты на основе уже ранее созданных объектов-прототипов. То есть по сути данный паттерн предлагает технику клонирования объектов.

ПРОТОТИП (PROTOTYPE)

Когда использовать Прототип?

- Когда конкретный тип создаваемого объекта должен определяться динамически во время выполнения

- Когда нежелательно создание отдельной иерархии классов фабрик для создания объектов-продуктов из параллельной иерархии классов (как это делается, например, при использовании паттерна Абстрактная фабрика)

- Когда клонирование объекта является более предпочтительным вариантом нежели его создание и инициализация с помощью конструктора. Особенно когда известно, что объект может принимать небольшое ограниченное число возможных состояний.

СТРОИТЕЛЬ (BUILDER)

Строитель (Builder) - шаблон проектирования, который инкапсулирует создание объекта и позволяет разделить его на различные этапы.

СТРОИТЕЛЬ (BUILDER)

Когда использовать паттерн Строитель?

- Когда процесс создания нового объекта не должен зависеть от того, из каких частей этот объект состоит и как эти части связаны между собой
- Когда необходимо обеспечить получение различных вариаций объекта в процессе его создания