Введение в Generic (обобщения, универсальные типы)

Generics

Что такое generics?

Необходимость использования generics.

Создание generic классов.

Сравнительный анализ generic классов.

Вложенные типы внутрь generic класса.

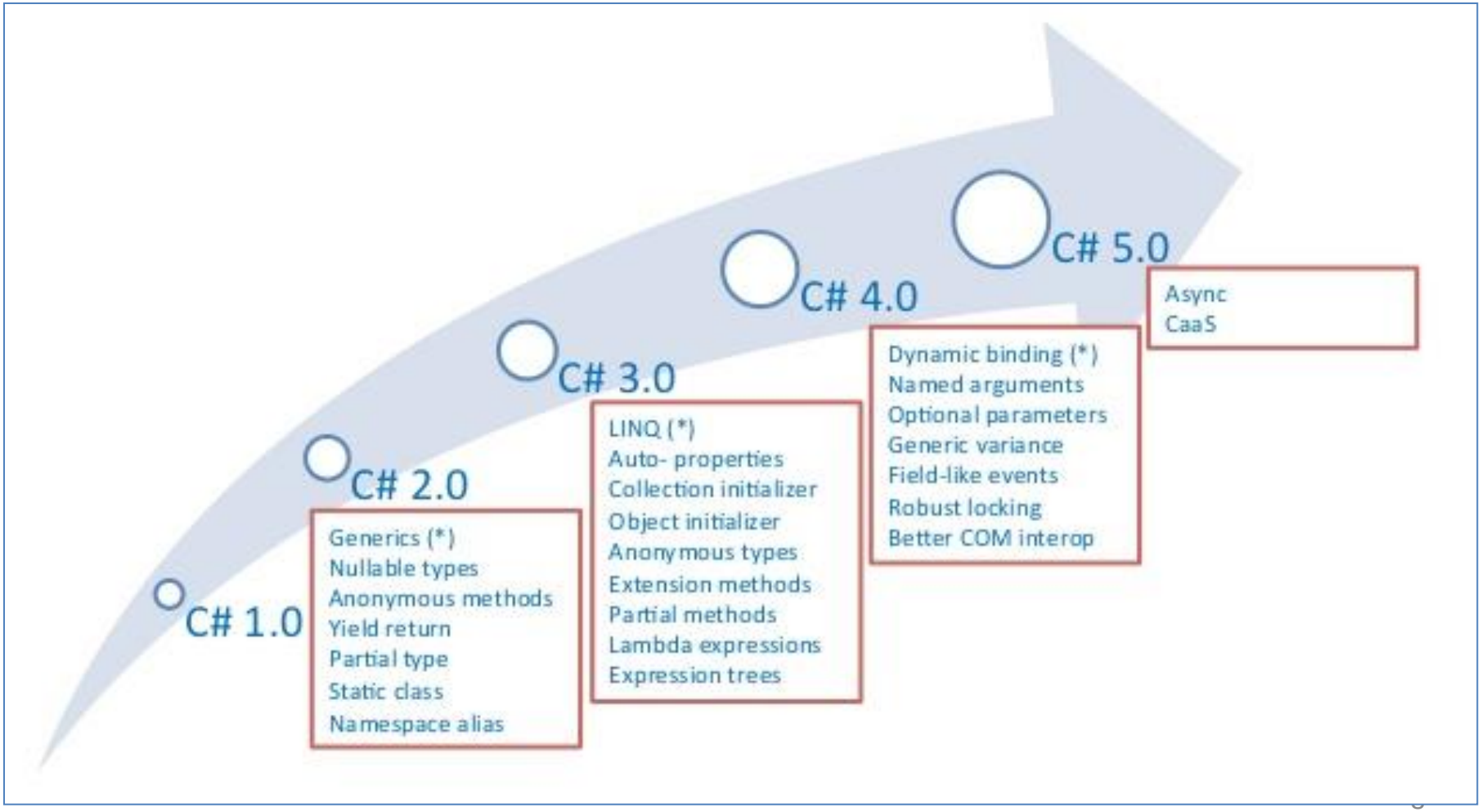
Использование ограничений.

Создание generic методов.

Создание generic интерфейсов.

Создание generic делегатов.

В **версии .NET 2.0** язык программирования C# был расширен поддержкой средства, которое называется **обобщением (generic)** и новым пространством имен, связанным с коллекциями, - **System.Collections.Generic**



Обобщение (Универсальные шаблоны) – элемент кода, способный адаптироваться для выполнения общих (сходных) действий над различными типами данных.

Обобщения создают параметризированный тип.

Особая роль параметризированных типов состоит в том, что они позволяют создавать **классы, структуры, интерфейсы, методы и делегаты**, в которых обрабатываемые данные **указываются в виде параметра**.

Обобщения —> шаблоны C++

Generic позволяют создавать обобщенные:

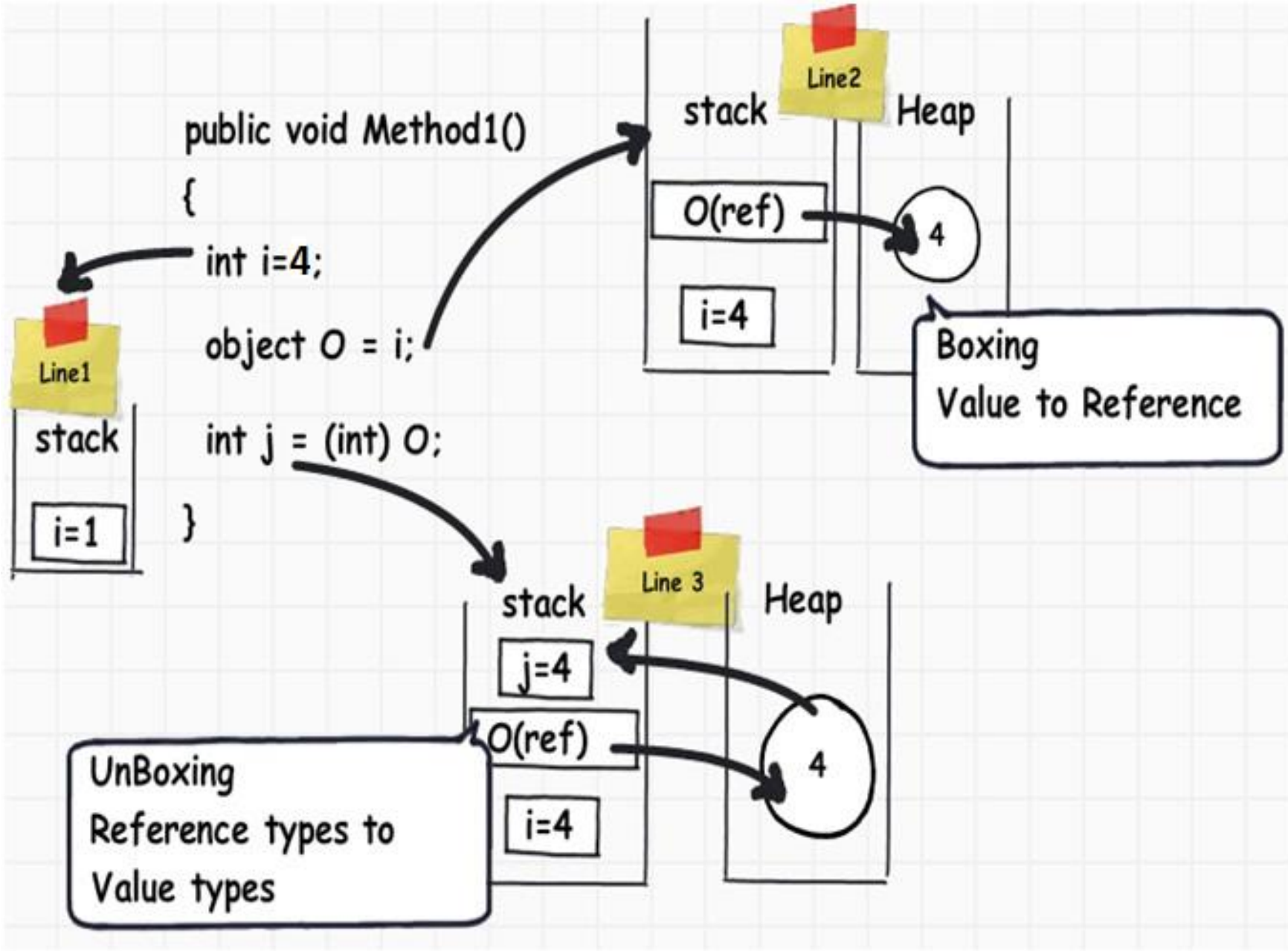
- классы
- структуры
- интерфейс
- делегаты
- методы

Параметры типов (<T>) используются для указания типов:

- полей класса
- параметров методов
- возвращаемых значений методов
- локальных переменных

Необходимость использования generics

- **необходимость указывать явное приведение типа** (может возникнуть ошибка в случае, если поместить в коллекцию переменную одного типа, а при извлечении выполнить приведение к другому типу)
- **снижение производительности** (выполнение упаковки и распаковки при хранении в коллекции переменных типов значений)


public void Method1()
{
int i=4;
object O = i;
int j = (int) O;
}
Line1
stack
i=1
Line2
stack
Heap
O(ref)
4
i=4
Boxing
Value to Reference
Line 3
stack
Heap
j=4
O(ref)
4
i=4
UnBoxing
Reference types to
Value types

Использование generic дает следующие преимущества:

- **безопасность типов** – в необобщенные коллекции можно было помещать любые объекты, в generic коллекцию можно поместить только объекты определенного типа (указанного при раскрытии шаблона);
- **более простой и понятный код**, т.к. ненужно выполнять приведение типа от object к конкретным типам;
- **повышение производительности** – при использовании generics структурные типы передаются по значению, упаковка и распаковка не происходит.

Пример создание generic классов

Общая форма объявления обобщенного класса:

```
class имя_класса<список_параметров_типа>
{
   // ...
}
```

Форма объявления ссылки на обобщенный класс:

```
имя_класса<список_аргументов_типа> имя_переменной =
     new имя_класса<список_пар-ов_типа>(список_арг-ов_конструктора);
```

Конструкция MyObj<T> называется ***открыто сконструированным типом***

Конструкция MyObj<int> называется ***закрыто сконструированным типом***

Создание generic классов

Обобщения позволяют создавать открытые **(open-ended)** типы, которые преобразуются в закрытые во время выполнения.

Идентификатор **<T>** – это указатель места заполнения, вместо которого подставляется любой тип.

```
class Types<T>
{
    T[] mass = new T[5];
}
```

Создание открытого типа

```
static void Main()
{
    Types<int> type = new Types<int>();
}
```

Закрытый тип

Создание параметризированного класса

Создаем объект и закрываем его типом int

Создание generic классов

- При **создании** generic класса параметр типа указывается в угловых скобках (< >) после имени класса. **Этот тип используется также как обычные типы**. Обобщенных параметров типа может быть несколько.

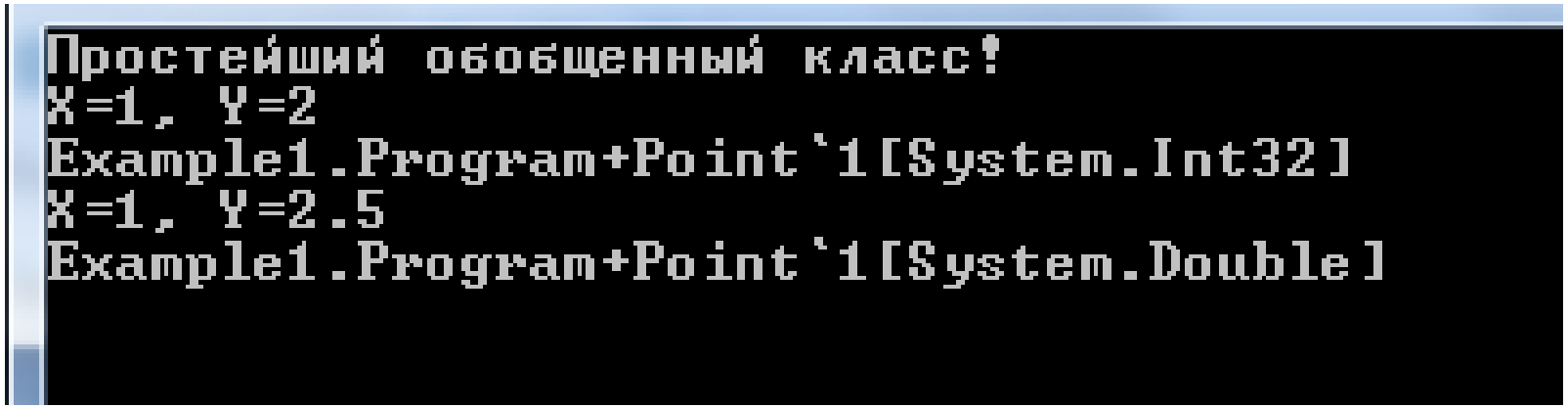
- При **использовании** generic класса вместо параметра типа подставляют **реальный тип данных (аргумент типа)**.

- Для задания значения по умолчанию переменным обобщенного типа используется выражение default(T). При этом значениям ссылочного типа присваивается null, а структурного 0.

Пример создание generic классов

```
public class Point<T>
 {
     T x;          // обобщенное поле
     public T X    // обобщенное свойство
      { get { return x; }  set { x = value; }  }

     T y;
     public T Y
      {  get { return y; } set { y = value; }  }

     public Point()          // конструктор без параметров
      {   this.x=default(T);  this.y=default(T);  }

     public Point(T x, T y) // конструктор
      {     this.x = x;   this.y = y;             }

     public override string ToString()
      {
           return String.Format("X={0}, Y={1}",
                                x.ToString(),
                                y.ToString());
       }
 }
```

```
Point<int> point1 = new Point<int>(1,2);
Console.WriteLine(point1);
Console.WriteLine(typeof(Point<int>).ToString());

Point<float> point2 = new Point<float>(1.0f, 2.5f);
Console.WriteLine(point2);
Console.WriteLine(typeof(Point<double>).ToString());
```



Перегрузка обобщенных типов

Перегрузки обобщенных типов различаются количеством параметров типа, а не их именами.

```
class MyClass<T>
{
    T[] mass = new T[5];
}
```

```
class MyClass<T,R>
{
    T[] mass = new T[5];
    R[] mass = new R[5];
}
```

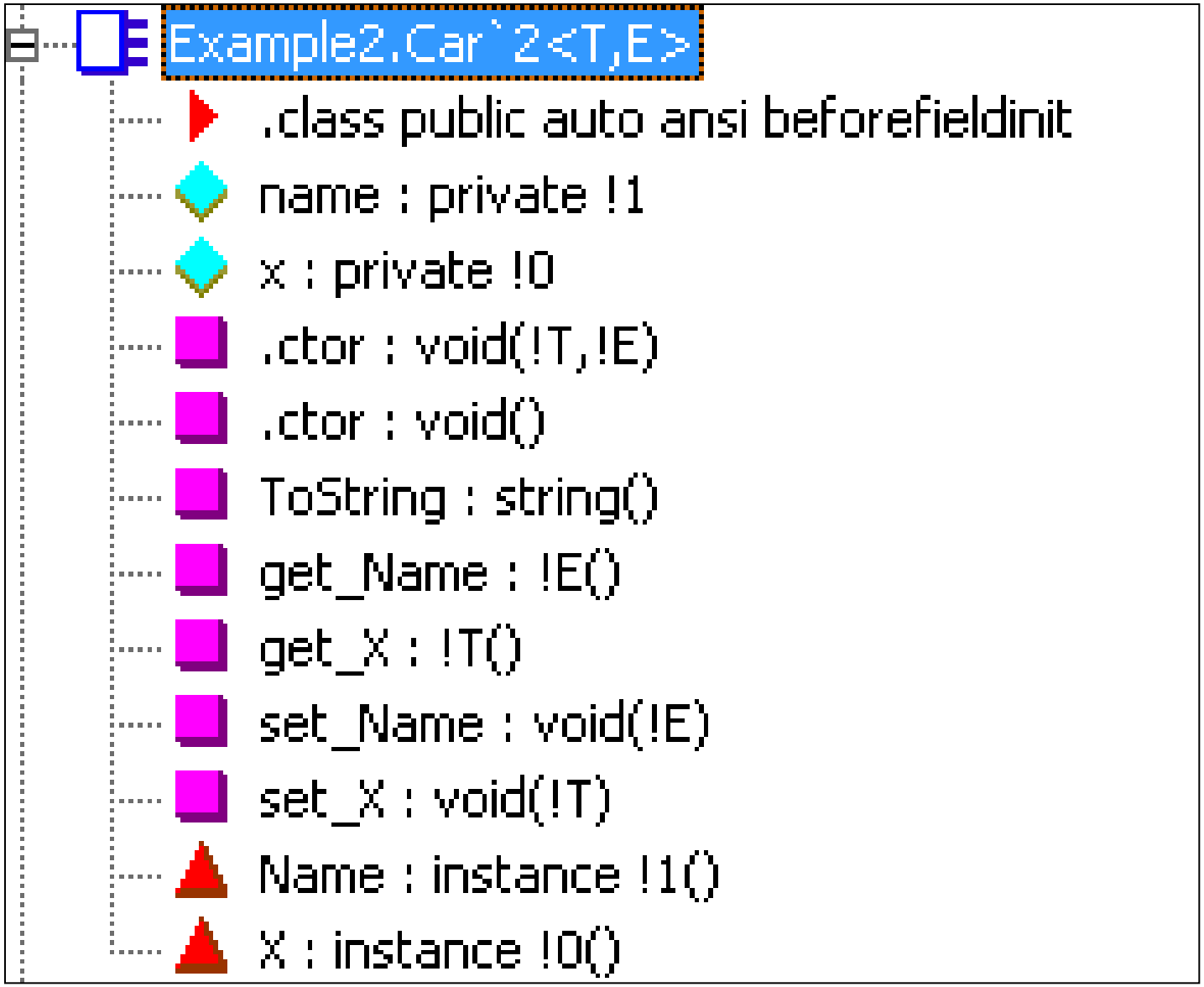
Пример создание generic классов

```
public class Car<T, E>
{
    T x;   // обобщенное поле
    public T X
    {  get { return x; }  set { x = value; }    }

    E name;
    public E Name
    {   get { return name; } set { name = value; }   }

    public Car() // конструктор
    {   this.x=default(T);    this.name=default(E);   }

    public Car(T x, E y) // конструктор
    {  this.x = x;    this.name = y;                  }

    public override string ToString()
    {
        return String.Format("№={0}, Имя={1}",
                             x.ToString(),
                             name.ToString());
    }
}
```

```
Car<int, string> car1 = new Car<int, string>(1,"BMV");
Console.WriteLine(car1);
Console.WriteLine(typeof(Car<int, string>).ToString());

Car<string, string> car2 = new Car<string, string>("1R852","BMV");
Console.WriteLine(car2);
Console.WriteLine(typeof(Car<string, string>).ToString());
```

```
Простейший обобщенный класс с двумя типами!
Номер=1, Наименование=BMV
Example2.Car`2[System.Int32,System.String]
Номер=1R852, Наименование=BMV
Example2.Car`2[System.String,System.String]
```

Вид созданного generic класса



Ограниченные типы

- ***Ограничение ссылочного типа***, требующее указывать аргумент ссылочного типа с помощью оператора **class**.
- ***Ограничение на базовый класс***, требующее наличия определенного базового класса в аргументе типа. Это ограничение накладывается указанием имени требуемого базового класса.
- ***Ограничение на интерфейс***, требующее реализации одного или нескольких интерфейсов аргументом типа. Это ограничение накладывается указанием имени требуемого интерфейса.
- ***Ограничение на конструктор***, требующее предоставить конструктор без параметров в аргументе типа. Это ограничение накладывается с помощью оператора **new ().**
- ***Ограничение типа значения***, требующее указывать аргумент типа значения с помощью оператора **struct**.

Список возможных ограничений:

Ограничение обобщения	Описание
where T: struct	Параметр типа должен наследоваться от system.ValueType, т.е. быть структурным типом
Where T: class	Параметр типа не должен наследоваться от system.ValueType, т.е. быть ссылочным типом
where T: new()	Класс должен иметь конструктор по умолчанию (указывается последним)
where T: BaseClass	Параметр типа должен быть производным классом от указанного базового класса
where T: Interface	Параметр типа должен реализовать указанный интерфейс

Синтаксис задания ограничения:

```
class ИмяКласс<T> where T: ограничения
```

Ограничение ссылочного типа

```
// указание ограничения - параметр типа должен быть значимым типом
public class Point<T> where T: struct
{
    T x;   // обобщенное поле
    public T X
    {  get { return x; }   set { x = value; }   }
    T y;
    public T Y
    {   get { return y; }  set { y = value; }   }

    public Point() // конструктор
    {   this.x=default(T);    this.y=default(T);    }

    public Point(T x, T y) // конструктор
    {   this.x = x;   this.y = y;                   }

    public override string ToString()
    {
    return String.Format("X={0}, Y={1}",x.ToString(), y.ToString());
    }
}
```

Ограничение ссылочного типа

```
Console.WriteLine("Использование ограничений в обобщенных классах");

Console.WriteLine("Простейший обобщенный класс!");
Point<int> point1 = new Point<int>(1,2);
Console.WriteLine(point1);
Console.WriteLine(typeof(Point<int>).ToString());

Point<float> point2 = new Point<float>(1.0f, 2.5f);
Console.WriteLine(point2);
Console.WriteLine(typeof(Point<double>).ToString());
Console.ReadKey();

// ОШИБКА ПАРАМЕТР ТИПА НЕ МОЖЕТ БЫТЬ ТИПОМ ССЫЛКИ
// Point<string> point2 = new Point<string>("1", "2");
```

Ограничение ссылочного типа и значимого типов

```
class ValSample<T> where T : struct
```

```
ValSample<int>
ValSample<FileMode>
```

```
ValSample<object>
ValSample<StringBuilder>
```

```
struct RefSample<T> where T : class
```

```
RefSample<IDisposable>
RefSample<string>
RefSample<int[]>
```

```
RefSample<Guid>
RefSample<int>
```

Ограничение на базовый класс

```
// Базовый класс, в котором
// хранятся имя абонента и номер его телефона,
    class PhoneNumber
    {
        public string Number { get; set; }
        public string Name { get; set; }
        public PhoneNumber(string n, string num)
        {
            Name = n;   Number = num;
        }
    }

    // Класс для телефонных номеров друзей
    class Friend : PhoneNumber
    {
       public bool IsWorkNumber { get; private set; }
       public Friend(string n, string num, bool wk) :  base (n, num)
       {   IsWorkNumber = wk;   }
    }
```

Ограничение на базовый класс

```
class PhoneList<T> where T:PhoneNumber
    {   T[] phList;
        int end;
        public PhoneList()
        {  phList = new T[10];  end = 0;
        }
        // Добавить элемент в список
        public bool Add(T newEntry)
        {  if (end == 10) return false;
            phList[end] = newEntry;
            end++;
            return true;
        }
        // Найти и возвратить сведения о телефоне по имени
        public T FindByName(string name)
        { for(int i=0; i<end; i++)
           {  if(phList[i].Name == name)
               return phList[i];
           }
          return null;
 } }
```

Ограничение на базовый класс

```
class Skype
    {
        public string Number { get; set; }
        public string Name { get; set; }
    }
```

```
Main()
{
   PhoneList<Friend> pList = new PhoneList<Friend>();
   pList.Add(new Friend("Иван", "", true));
   pList.Add(new Friend("Петр", "", true));
   pList.Add(new Friend("Илья", "", true));
   try
     {   Friend frnd = pList.FindByName("Петр");
         Console.Write(frnd.Name + ": " + frnd.Number);
         if(frnd.IsWorkNumber)
           Console.WriteLine(" (рабочий)") ;
     }
  catch  {  Console.WriteLine("Не найдено");  }

  // Ошибка - т.к. класс Skype - не наследует PhoneNumber
  // PhoneList<Skype> skList = new PhoneList<Skype>();
}
```

Ограничение на конструктор

```
// класс, имеющий конструктор по умолчанию
   class A
   {  public int ID { get; set; }  }
   // класс, не имеющий конструктор по умолчанию
   class B
   {   public string  Name { get; set; }
       public B(string name)
       {  Name = name;    }
   }
   // обобщенный класс, демонстрирующий ограничения на конструктор
   class Test<T> where T: new ()
   {  public Test()
        {

        }
   }
```

```
Test<A> objA = new Test<A>();
// Ошибка параметр типа не содержит конструктор по умолчанию
//Test<B> ObjB = new Test<B>();
```

Ограничение на конструктор

```
public T CreateInstance<T>() where T : new()
{
        return new T();
}
```

```
CreateInstance<int>()
CreateInstance<object>()
```

```
CreateInstance<string>()
```

Добавление ограничений для обобщенных типов. Ограничения преобразования типа

Declaration	Constructed type examples
`class Sample<T> where T : Stream`	*Valid:* `Sample<Stream>` (identity conversion) *Invalid:* `Sample<string>`
`struct Sample<T> where T : IDisposable`	*Valid:* `Sample<SqlConnection>` (reference conversion) *Invalid:* `Sample<StringBuilder>`
`class Sample<T> where T : IComparable<T>`	*Valid:* `Sample<int>` (boxing conversion) *Invalid:* `Sample<FileInfo>`
`class Sample<T,U> where T : U`	*Valid:* `Sample<Stream,IDisposable>` (reference conversion) *Invalid:* `Sample<string,IDisposable>`

Добавление ограничений для обобщенных типов

```
class Sample<T> where T : class, IDisposable, new()
class Sample<T> where T : struct, IDisposable
class Sample<T,U> where T : class where U : struct, T
class Sample<T,U> where T : Stream where U : IDisposable
```

```
class Sample<T> where T : class, struct
class Sample<T> where T : Stream, class
class Sample<T> where T : new(), Stream
class Sample<T> where T : IDisposable, Stream
class Sample<T> where T : XmlReader, IComparable, IComparable
class Sample<T,U> where T : struct where U : class, T
class Sample<T,U> where T : Stream, U : IDisposable
```

Обобщенный метод

- Параметр типа объявляется **после имени метода**, **но** **перед списком его параметров**.
- Обобщённые методы могут быть **статическими**, что позволяет вызывать их **независимо от любого объекта**.
- Обобщенные методы могут вызывать как обычные методы – **без указания аргументов типа**. Этот процесс называется выводимостью типов.
- В обобщенных методах также могут **применятся ограничения**.

Обобщенный метод

```
class MyClass
{
  public static void Swap<T> (ref T a, ref T b)
  {  Console.WriteLine("Передан в метод Swap() тип {0}", typeof(T));
     T temp;
     temp = a;
     a = b;
     b = temp;
}}
```

```
// Обмен  двух значений
int  a = 10, b = 90;
Console.WriteLine("Перед: {0}, {1}", a, b);
MyClass.Swap<int>(ref a, ref b);
Console.WriteLine("После: {0}, {1}", a, b);
Console.WriteLine();

// Обмен  двух строк.
string s1 = "Привет", s2 = "Пока";
Console.WriteLine("Перед: {0} {1}!", s1, s2);
MyClass.Swap(ref s1, ref s2); // Параметр типа можно не указывать
Console.WriteLine("После: {0} {1}!", s1, s2);
```

Обобщенный интерфейс

```
public interface IBinaryOperations<T>
{
        T Add(T arg1, T arg2);
        T Subtract(T arg1, T arg2);
        T Multiply(T arg1, T arg2);
        T Divide(T arg1, T arg2);
}
```

```
class BasicMath : IBinaryOperations<int>
{       public int Add(int arg1, int arg2)
         { return arg1 + arg2; }

        public int Subtract(int arg1, int arg2)
         { return arg1 - arg2; }

        public int Multiply(int arg1, int arg2)
         { return arg1 * arg2; }

        public int Divide(int arg1, int arg2)
         { return arg1 / arg2; }
}
```

```
BasicMath m = new BasicMath();
Console.WriteLine("1 + 1 = {0}", m.Add(1, 1));
```

Определение и реализация ковариантного интерфейса

Если параметр типа в обобщенном интерфейсе появляется только в качестве **возвращаемого значения методов**, то можно сообщить компилятору, что некоторые неявные преобразования являются законными и что можно не соблюдать строгую безопасность типов

```
public interface IContainerGet<out T>
    {
        T GetItem();
    }
```

В обобщенном интерфейсе `IContainerGet<out T>` поддерживается ковариантность:

- ключевое слово **out** обозначает, что обобщенный тип T является ковариантным.
- метод **`GetItem()`** может возвращать ссылку на обобщенный тип T или ссылку на любой класс, производный от типа T.

Ограничения:

- Ковариантность параметра типа может распространяться только на **тип, возвращаемый методом**.
- Ковариантность оказывается пригодной только **для ссылочных типов**.
- Ковариантный тип нельзя использовать **в качестве ограничения в интерфейсном методе**.

Определение и реализация ковариантного интерфейса

```
public class AutoPark<T> : IContainerGet<T>, IContainerSet<T>
{
    private T item;

    public T GetItem()
    {
        return item;
    }

    public void SetItem(T value)
    {
     item=value;
    }
}
```

```
// обычное поведение
IContainerGet<Car> listCars = new AutoPark<Car>();
// обычное поведение
IContainerGet<Truck> listTrucks = new AutoPark<Truck>();
// невозможно было реализовать без out
IContainerGet<Car> listCarTrucks = new AutoPark<Truck>();
```

Определение и реализация контравариантного интерфейса

Если параметр типа в обобщенном интерфейсе появляется только в качестве **аргументов методов**, то можно сообщить компилятору, что некоторые неявные преобразования являются законными и что можно не соблюдать строгую безопасность типов

```
public interface IComparer<in T>
{
    int Compare(T x, T y);
}
```

```
delegate возвращаемый_тип
        имя_делегата<список_параметров_типа>{список_аргументов) ;
```

```
// Этот обобщенный делегат может вызвать любой метод,
// возвращающий void и принимающий один параметр.
public delegate void MyGenericDelegate<T>(T arg);
```

```
static void StringTarget(string arg)
{
   Console.WriteLine("arg в верхнем регистре: {0}", arg.ToUpper());
}

static void IntTarget(int arg)
{
            Console.WriteLine("++arg: {0}", ++arg);
}
```

```
MyGenericDelegate<string> strTarget =
                          new MyGenericDelegate<string>(StringTarget);
strTarget("Некоторые строковые данные");


MyGenericDelegate<int> intTarget = IntTarget;
intTarget(9);
```

Иерархии обобщенных классов

- **Обобщенные классы** могут входить в иерархию классов аналогично **необобщенным классам**.
- **Обобщенный класс** может действовать как **базовый** или **производный** класс.
- Обобщенные классы могут иметь **виртуальные и абстрактные методы**.

Главное отличие между иерархиями обобщенных и необобщенных классов:

аргументы типа, необходимые обобщенному базовому классу, должны **передаваться всеми производными классами вверх** по иерархии аналогично передаче аргументов конструктора.

Правила наследования от generic классов:

- если от **generic** класса наследуется необобщенный класс – **наследник должен конкретизировать параметр типа**
- при реализации **generic виртуальных методов** производный необобщенный класс должен **конкретизировать параметр типа**
- если от **generic класса наследуется другой generic класс**, в нем необходимо **учитывать ограничения** типа, указанные в базовом классе.

Пример наследования generic классов

```
public class Point<T>
{
        T x;   // обобщенное поле
        public T X
        {   get { return x; }  set { x = value; }   }
        T y;
        public T Y
        {   get { return y; }  set { y = value; }   }
        public Point() // конструктор
        {
            this.x = default(T);
            this.y = default(T);
        }
        public Point(T x, T y) // конструктор
        {   this.x = x;
            this.y = y;
        }
 public virtual string GetInfo()
 {
  return String.Format("X={0}, Y={1}", x.ToString(), y.ToString());
 }
}
```

```
public class ColorPoint1<T> : Point<T>
    {
        string  Color { get; set; }

        public ColorPoint1(string color, T x, T y)
            : base(x, y)
        {
            Color = color;
        }

        public override string GetInfo()
        {
            return base.GetInfo() + "Цвет: " + Color;
        }
    }
```

```
Console.WriteLine("Наследование обобщенных классов");
ColorPoint1<int> cPoint1 = new ColorPoint1<int>("Red", 10, 20);
Console.WriteLine(cPoint1.GetInfo());
```

```
// наследование от обобщенного класса
 public class ColorPoint1<W, T>:Point<T>
{
    W Color {get; set;}

    public ColorPoint1 (W color,T x , T y):base (x,y)
    {
      Color =color;
    }

    public override string GetInfo()
    {
      return base.GetInfo()+"Цвет: "+Color;
    }
}
```

```
ColorPoint1<string, int> cPoint1 =
                          new ColorPoint1<string, int>("Red", 10,20);
Console.WriteLine(cPoint1.GetInfo());
```

```
public class ColorPoint2<W>:Point<short>
{
    W Color {get; set;}

    public ColorPoint2 (W color, short x , short y):base(x, y)
    {
       Color =color;
    }

    public override string GetInfo()
    {
       return base.GetInfo()+"Цвет: "+Color;
    }
}
```

```
ColorPoint2<string> cPoint2 =
                              new ColorPoint2<string>("Green", 12,17);
Console.WriteLine(cPoint2.GetInfo());
```

Общие сведения об универсальных шаблонах:

- Используйте универсальные типы для достижения максимального уровня повторного использования кода, безопасности типа и производительности.
- Наиболее частым случаем использования универсальных шаблонов является создание классов коллекции.
- Можно создавать собственные универсальные интерфейсы, классы, методы, события и делегаты.
- Доступ универсальных классов к методам можно ограничить определенными типами данных

Тип Nullable

Тип Nullable<T> представляет типы значений с пустыми (нулевыми) значениями.

```
int? a = null;
int? b = a + 4;
```

b = null

При сравнении операндов один из которых `null` - результатом сравнения всегда будет - `false`

```
int? a = null;
int? b = -5;

if (a >= b)  // - false
{
    Console.WriteLine("a >= b");
}
else
{
    Console.WriteLine("a < b");
}
```

Операция поглощения

```
static void Main()
{
    int? a = null;
    int? b;

    b = a ?? 10; // b = 10

    a = 3;
    b = a ?? 10; // b = 3
}
```

Оператор **??** возвращает левый операнд, если он не `null` и правый операнд, если левый `null`.

Кортеж — особый тип структуры данных

Кортеж — это некоторая группа объектов (переменных, констант), не имеющая собственного типа, а существующая на этапе компиляции просто для удобства.

Массивы комбинируют объекты одного типа, а ***кортежи (tuple)*** могут комбинировать объекты различных типов.

Понятие кортежей происходит из языков функционального программирования, таких как F#, где они часто используются. С появлением .NET 4 кортежи стали доступны в .NET Framework для всех языков .NET.

В .NET 4 определены восемь обобщенных классов Tuple и один статический класс Tuple, который служит фабрикой кортежей.
Существуют различные обобщенные классы Tuple для поддержки различного количества элементов:
например, Tuple<T1> содержит один элемент, Tuple<T1, T2> — два элемента и т.д.

```
// Данный метод возвращает кортеж с 4-мя  разными значениями
static Tuple<int, double, string, char> GenTuple(int a, string name)
{
    int sqr = a * a;
    double sqrt = Math.Sqrt(a);
    string s = "Привет, " + name;
    char ch = (char)(name[0]);
    return Tuple.Create<int, double, string, char>(sqr, sqrt, s, ch);
}

static void Main(string[] args)
{
 var myTuple = GenTuple (25, "Alexandr");

 Console.WriteLine("{0}\n25 в квадрате: {1}\nКвадратный корень из
                    25:{2}\nПервый символ в имени: {3}\n",
                                    myTuple.Item3,
                                    myTuple.Item1,
                                    myTuple.Item2,
                                    myTuple.Item4);
}
```

```
Привет, Alexandr
25 в квадрате: 625
Квадратный корень из 25:5
Первый символ в имени: A
```