Лекция № 1

I T E A

IT Education Academy

WWW.ITEA.UA

C# Advanced

Урок № 1

Обобщения

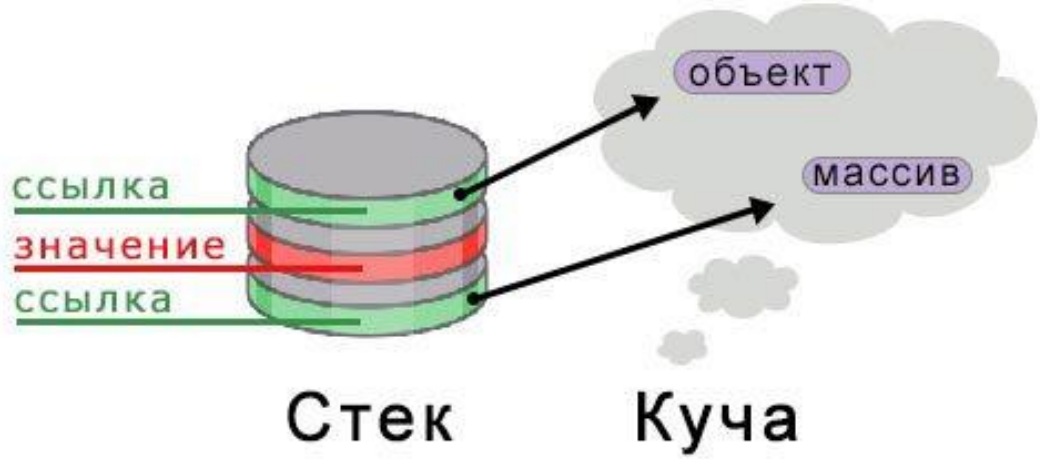
План урока

- Устройство памяти в .NET
- Упаковка (boxing) и распаковка (unboxing)
- Причина возникновения обобщенных типов
- Обобщения
- Общие сведения об универсальных шаблонах
- Ковариантность и контрвариантность обобщений
- Ограничения в обобщениях
- Обобщенные интерфейсы
- Тип Nullable

Устройство памяти в .NET

Стек — это область оперативной памяти, которая создаётся для каждого потока. Он работает в порядке LIFO (Last In, First Out), то есть последний добавленный в стек кусок памяти будет первым в очереди на вывод из стека.

Куча — это хранилище памяти, также расположенное в ОЗУ, которое допускает динамическое выделение памяти и не работает по принципу стека: это просто склад для ваших переменных. Когда вы выделяете в куче участок памяти для хранения переменной, к ней можно обратиться не только в потоке, но и во всем приложении



Понятие класса и объекта

Система типов NET

Хранение - Стек

Хранение - Куча

Типы значений

Целочисленные типы
Типы с плавающей запятой
Тип decimal
Тип char
Перечисления enum
Структуры (struct)

Ссылочные типы

Тип object
Тип string
Классы (class)
Интерфейсы (interface)
Делегаты (delegate)

Создание объектов с помощью классов

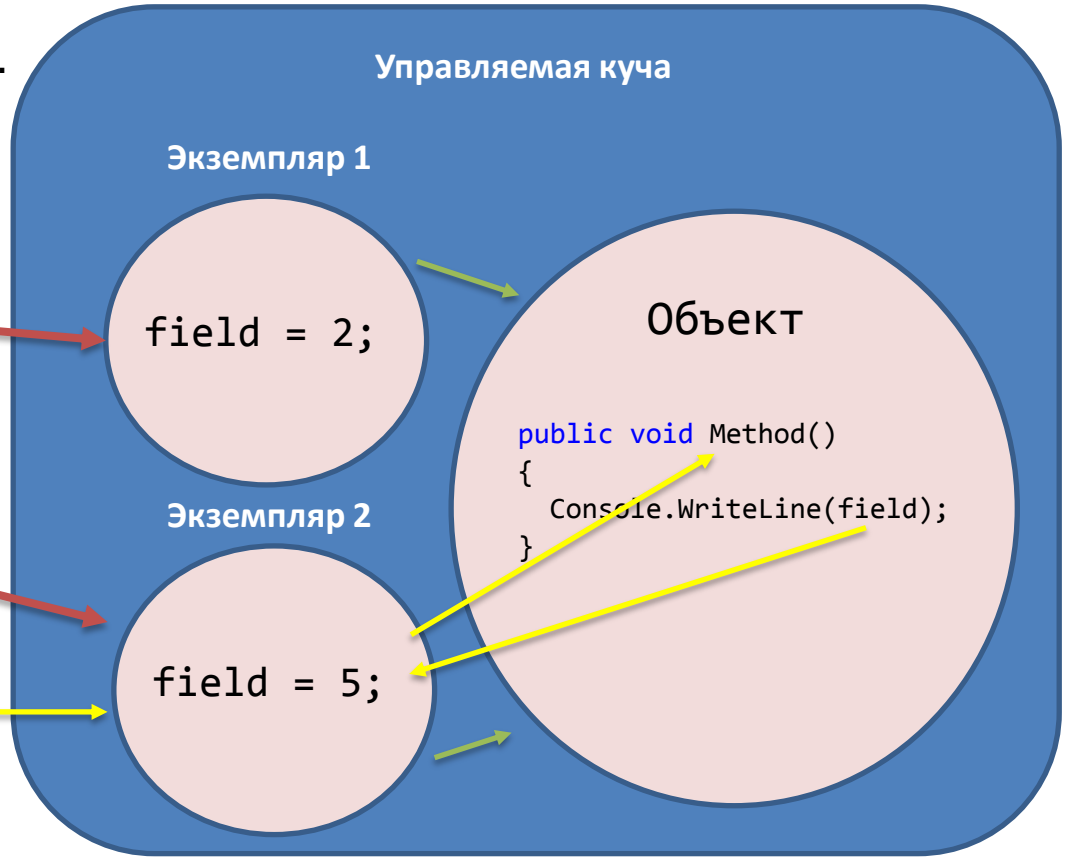
Класс и объект — это разные вещи. **Класс** определяет тип объекта, но не сам объект.

Объект — это конкретная сущность, основанная на классе и иногда называемая экземпляром класса.

Объекты содержат в себе статические поля и все методы.

Экземпляры содержат нестатические поля.

```
MyClass instance1 = new MyClass();
MyClass instance2 = new MyClass();

instance1.field = 2;
instance2.field = 5;

instance1.Method();
instance2.Method();
```



Упаковка-Распаковка (Boxing/UnBoxing)

- Когда любой значимый тип присваивается к ссылочному типу данных, значение перемещается из области стека в кучу. Эта операция называется упаковкой.

- Когда любой ссылочный тип присваивается к значимому типу данных, значение перемещается из области кучи в стек. Это называется распаковкой.

- Упаковка является неявной, распаковка же является явной.

- Значимые типы легче ссылочных.

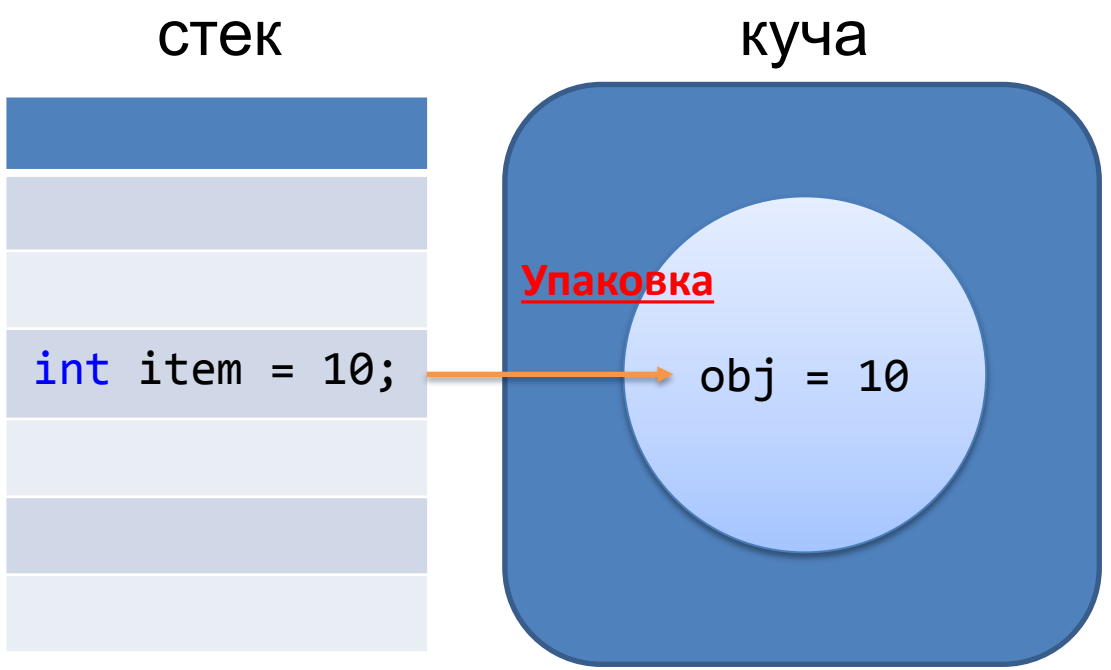
Упаковка (Boxing)

Упаковка это преобразование value (значимого) типа в Object тип, преобразование является неявным.

```
static void Main()
{
    //Значимый тип
    int item = 10;

    //Ссылочный тип
    object obj = item;
}
```

стек

куча



Распаковка (UnBoxing)

Распаковка это преобразование Object типа в value (значимый) тип.

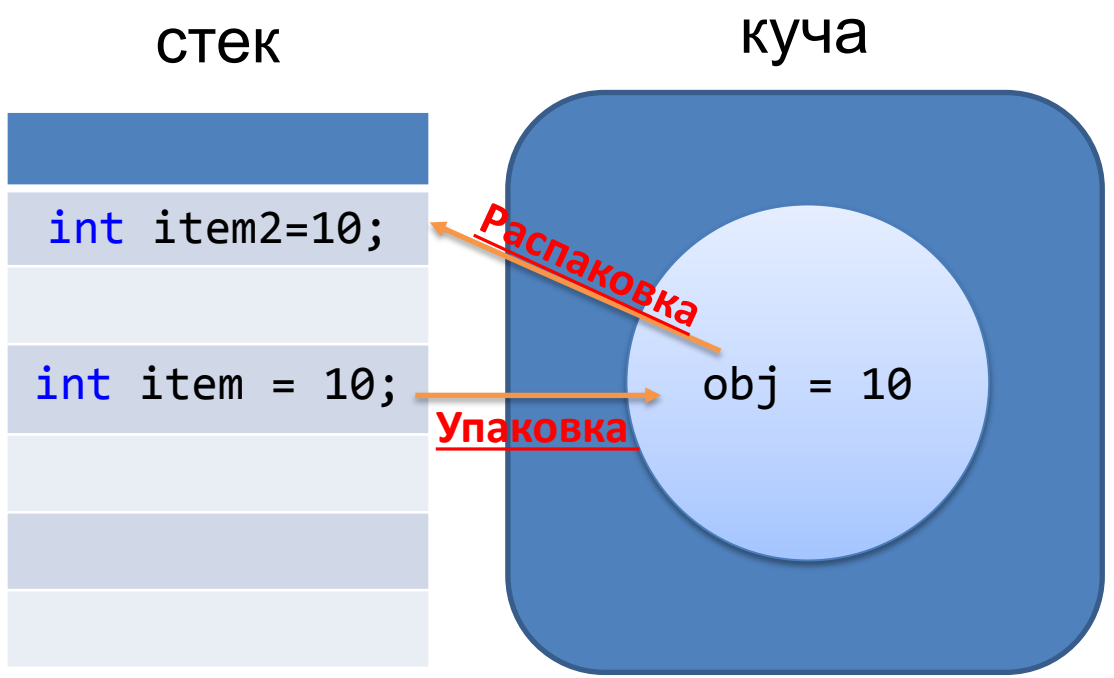
```
static void Main()
{
    //Значимый тип
    int item = 10;

    //Ссылочный тип
    object obj = item;

    //Значимый тип
    int item2 = (int)obj;
}
```

стек

куча

Упаковка

Распаковка



Распаковка (UnBoxing)

Перечень действий, которые должны быть выполнены при упаковке и распаковке простого целого числа.

1. Новый объект должен быть размещен в управляемой куче.
2. Значение данных, находящихся в стеке, должно быть передано в выделенное место в памяти.
3. При распаковке значение, которое хранится в объекте, находящемся в куче, должно быть передано обратно в стек.
4. Неиспользуемый больше объект в куче будет (в конечном итоге) удален сборщиком мусора.

=> проблемы с производительность.

Причина возникновения обобщенных типов

Проблема, которая могла возникнуть до появления обобщенных типов.

Например, мы определяем класс для представления сотрудника:

```
class Employee
{
   public int PersonnelNumber { get; set; } // Табельный номер (уникальный идентификатор )
   public string Surname { get; set; }      // Фамилия
}
```

Вопрос: Если на момент написания класса мы можем точно не знать, какой тип данных выбрать для хранения идентификатора (табельного номера) - строку или число.

Причина возникновения обобщенных типов

Первый вариант решения проблемы - мы можем определить свойство PersonnelNumber как свойство типа object, так как object является универсальным типом, то мы сохранить и строку, и число.

```
public object PersonnelNumber { get; set; }
```

При таком подходе все вроде работает, но такое решение является не оптимальным, так как, в данном случае мы сталкиваемся с такими явлениями как **упаковка (boxing)** и **распаковка (unboxing).**

```
static void Main()
{
   Employee employee = new Employee();                    // создание экземпляров класса Employee
   employee. PersonnelNumber = 1;                            // упаковка в значения int в тип Object
   int personnelNumber = (int)employee2.PersonnelNumber;  // распаковка в тип int
}
```

Причина возникновения обобщенных типов

Кроме того, существует другая проблема - **проблема безопасности типов**. Так, мы получим ошибку во время выполнения программы, если поступим следующим образом:

```
static void Main()
{
    Employee employee = new Employee(); // создание экземпляров класса Employee
    employee.PersonnelNumber = "123";
    int personnelNumber = (int)employee.PersonnelNumber; // Исключение InvalidCastException
}
```

Мы можем не знать, какой именно объект представляет PersonnelNumber, и при попытке получить число в данном случае мы столкнемся с исключением InvalidCastException.

Обобщение

Все выше описанные проблемы были призваны устранить **обобщенные типы** или как их часто называют **универсальные шаблоны (типы)**.

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

Общая форма объявления обобщенного класса:

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

Например:

```
class MyClass<T>{}
```

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

Обобщение

Класс Employee как обобщённый:

```
class Employee<T>
{
   public T PersonnelNumber { get; set; }
   public string Surname { get; set; }
}
```

T - универсальный параметр, так как вместо него можно подставить любой тип.

Вместо параметра T можно использовать объект int, то есть число, представляющее табельный номер сотрудника, также это может быть объект string, или любой другой класс или структура.

```
static void Main()
{
   Employee<string> employee1 = new Employee<string>() ;
   Employee<int> employee2 = new Employee<int>();
}
```

Обобщение

- Обобщения позволяют создавать открытые (open-ended) типы, которые преобразуются в закрытые вовремя выполнения.
- Каждый закрытый тип получает свою собственную копию набора статических полей.
- Идентификатор <T> – это указатель места заполнения, вместо которого подставляется любой тип.

Создание открытого типа

```
class MyClass<T>
{
   T[] array = new T[10];
}
```

Закрытый тип

```
static void Main()
{
   MyClass<string> myClass = new MyClass<string>();
}
```

Перегрузка обобщенных типов

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

Класс с универсальный параметром T

```
class MyClass<T>
{
   T[] array = new T[10];
}
```

Класс с универсальными параметрами T и U

```
class MyClass<T, U>
{
   T[] array1 = new T[10];
   U[] array2 = new U[10];
}
```

```
static void Main()
{
   MyClass<int> myClass1 = new MyClass<int>();
   MyClass<int, string> myClass2 = new MyClass<int, string>();
}
```

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

Ковариантность обобщений

- **Ковариантность**: позволяет использовать более конкретный тип, чем заданный изначально.
- **Ковариантность** обобщений – UpCast параметров типов.
- **Ковариантность** обобщений в C# 4.0 ограничена интерфейсами и делегатами.

```
public class Shape { }
public class Circle : Shape { }
public interface IContainer<out T> {}
public class Container<T> : IContainer<T>{}

static void Main()
{
   Circle circle = new Circle();
   IContainer<Shape> container = new Container<Circle>();
}
```

Обобщенные интерфейсы и делегаты могут быть **ковариантными**, если к универсальному параметру применяется ключевое слово **out**.

Контрвариантность обобщений

- **Контрвариантность** позволяет использовать более универсальный тип, чем заданный изначально.
- **Контрвариантность** обобщений – DownCast параметров типов.
- **Контрвариантность** обобщений в C# 4.0 ограничена интерфейсами и делегатами.

```
public class Shape { }
public class Circle : Shape { }
public interface IContainer<in T> {}
public class Container<T> : IContainer<T>{}

static void Main()
{
   Circle circle = new Circle();
   IContainer<Circle> container = new Container<Shape>();
}
```

Для создания контравариантного интерфейса или делегата надо использовать ключевое слово in.

Ограничения в обобщениях

Ограничение where

Предложение where используется в определении универсального типа для указания ограничений типов, которые могут использоваться в качестве аргументов параметра типа, определенного в универсальном объявлении.

- Ограничение, чтобы использовались только структуры или другие типы значений:

```
class MyClass<T> where T: struct {}
```

- Ограничение, чтобы использовались только ссылочные типы:

```
class MyClass<T> where T: class { }
```

Ограничения в обобщениях

Ограничение new

Ограничение new() указывает, что аргумент любого типа в объявлении общего класса должен иметь открытый конструктор без параметров.

```
class MyClass<T> where T : new()
{
   public T instance = new T();
}
```

new() - универсальный параметр должен представлять тип, который имеет общедоступный (public) конструктор без параметров.

Ограничения в обобщениях

Правила использования ограничений

where T : struct – Аргумент типа должен быть структурного типа, кроме Nullable.

where T : class – Аргумент типа должен иметь ссылочный тип; это также распространяется на тип любого класса, интерфейса, делегата или массива.

where T : <base class name> - Аргумент типа должен являться или быть производным от указанного базового класса

where T : U – Аргумент типа, поставляемый для T, должен являться или быть производным от аргумента, поставляемого для U. Это называется неприкрытым ограничением типа

Тип Nullable

- Значение **null** по умолчанию могут принимать только объекты ссылочных типов.
- Однако в различных ситуациях бывает удобно, чтобы объекты числовых типов данных имели значение **null**, то есть были бы не определены.
- Для этого надо использовать знак вопроса **?** после типа значений.

```
int? variable = null;
bool? predicate = null;
```

- Фактически запись ? является упрощенной формой использования структуры **System.Nullable<T>**.

```
Nullable<int> variable = null;
Nullable<bool> predicate = null;
```

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

Тип Nullable

```
static void Main()
{
   int? a = null;
   int? b = -5;
   if (a >= b) // false
      Console.WriteLine("a >= b");
   else
      Console.WriteLine("a < b");
}
```

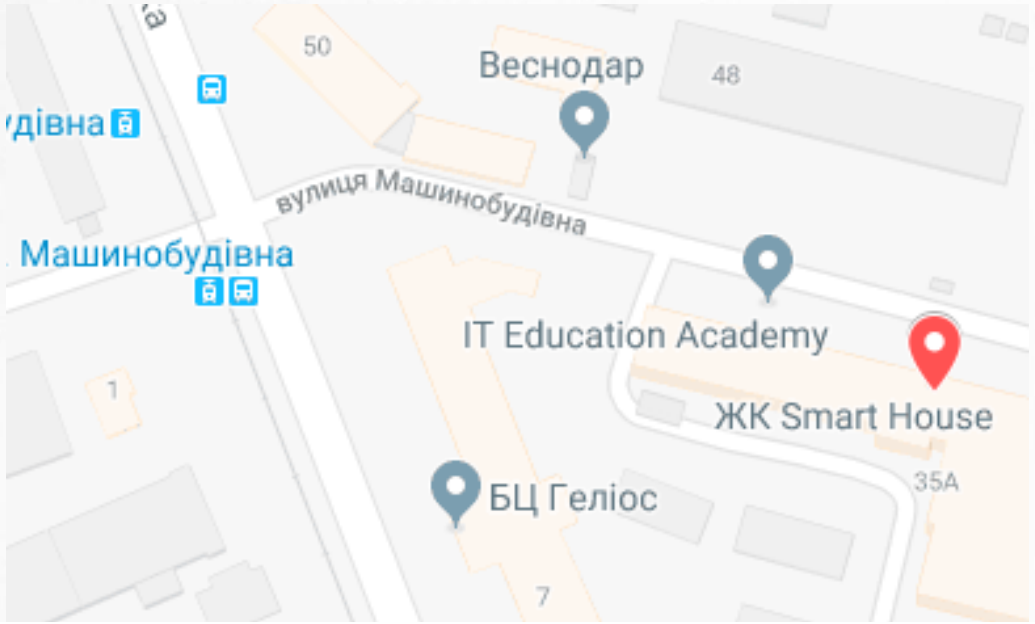
При сравнении операндов один из которых null-результатом сравнения всегда будет – false.

Операция поглощения

Оператор ?? Возвращает левый операнд, если он не null и правый операнд, если левый null.

```
static void Main()
{
   int? a = null;
   int? b;
   b = a ?? 10; // b = 10
   a = 3;
   b = a ?? 10; // b = 3
}
```


Презентация окончена.
Спасибо за внимание!



I T E A

КОНТАКТНЫЕ ДАННЫЕ

ITEA

ЖК “Smart House”, ул. Машиностроительная, 41 (м.Берестейская)

ЖК «Корона» улица Срибнокильская,1 м. Позняки

+38 (044) 599-01-79
facebook.com/Itea
info@itea.ua
itea.ua