Alex_BBB сегодня в 18:22

Шпаргалка для собеседования .Net

.NET, C#, Карьера в IT-индустрии

Ниже не учебник, а только шпаргалка для разработчиков уже знакомых с основами C# .Net.

Шпаргалка содержит только вопросы "на базу". Вопросы вида "как бы вы спроектировали ...", "какие слои приложения ...", в шпаргалку не входят. Как отметили в комментариях, вопросы скорее для джуна, темнее менее их задают и на собеседованиях миддлов.

▾ Форматирование кода

В примерах, для краткости, открывающая скобочка { не на новой строке. Интервьюер может быть смущен, т.к. в C# принято ставить { с новой строки. Поэтому на собеседовании лучше использовать общепринятое форматирование.

stack и heap, value type и rerence type

- refernce type (пример class, interface) хранятся в большой heap
- value type (пример int, struct, ссылки на инстансы refrence type) хранятся в быстром stack
- при присвоении (передачи в метод) value type копируются, reference type передаются по ссылке (см. ниже раздел struct)

struct

- value type => при присвоении (передачи в метод) все поля и свойства копируются, не может быть null
- нет наследования
- поддерживает интерфейсы
- если есть конструктор, в нем должны устанавливаться все поля и свойства

```
interface IMyInterface {
    int Property { get; set; }
}

struct MyStruc : IMyInterface {
    public int Field;
    public int Property { get; set; }
}

class Program {
    static void Main(string[] args) {
        var ms = new MyStruc {
            Field = 1,
            Property = 2
        };
        // при передаче в метод value type копируется,
        // поэтому в методе будет другая переменная
        TryChange(ms);
        Console.WriteLine(ms.Property);
        // ==> ms.Property = 2;

        // тут происходит boxing (см ниже)
        IMyInterface msInterface = new MyStruc {
            Field = 1,
            Property = 2
        };
```

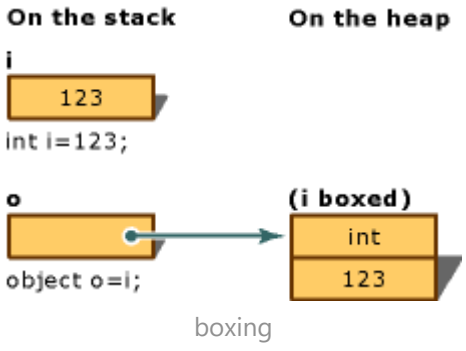
```
            // поэтому в метод передается уже object (reference type)
            // внутри метода будет не другая переменная, а ссылка на msInterface
            TryChange(msInterface);
            Console.WriteLine(msInterface.Property);
            // ==> ms.Property = 3;
        }

        static void TryChange(IMyInterface myStruc) {
            myStruc.Property = 3;
        }
    }
```

DateTime это struct, поэтому проверять поля типа DateTime на null бессмысленно:

```
class MyClass {
    public DateTime Date { get; set; }
}

var mc = new MyClass();
// всегда false,
// т.к. DateTime это struct (value type) не может быть null
var isDate = mc.Date == null;
```

boxing / unboxing

```
// boxing (value type, stack -> object, heap)
int i = 123;
object o = i;

// unboxing (object, heap -> value type, stack)
object o = 123;
var i = (int)o;
```



boxing

```
// пример boxing
int i = 123;
object o = i;
i = 456;
// результат ==> т.к. i, o хранятся в разных ячейках памяти
//  i = 456
//  o = 123
```

Зачем это нужно

```
// При приведении структуры к интерфейсу происходит boxing
IMyInterface myInterface = new MyStruct(2);
```

```
// boxing i
int i = 2;
string s = "str" + i;
// т.к. это String.Concat(object? arg0, object? arg1)

// unboxing, т.к. Session Dictionary<string, object>
int i = (int)Session["key"];
```

string особенный тип

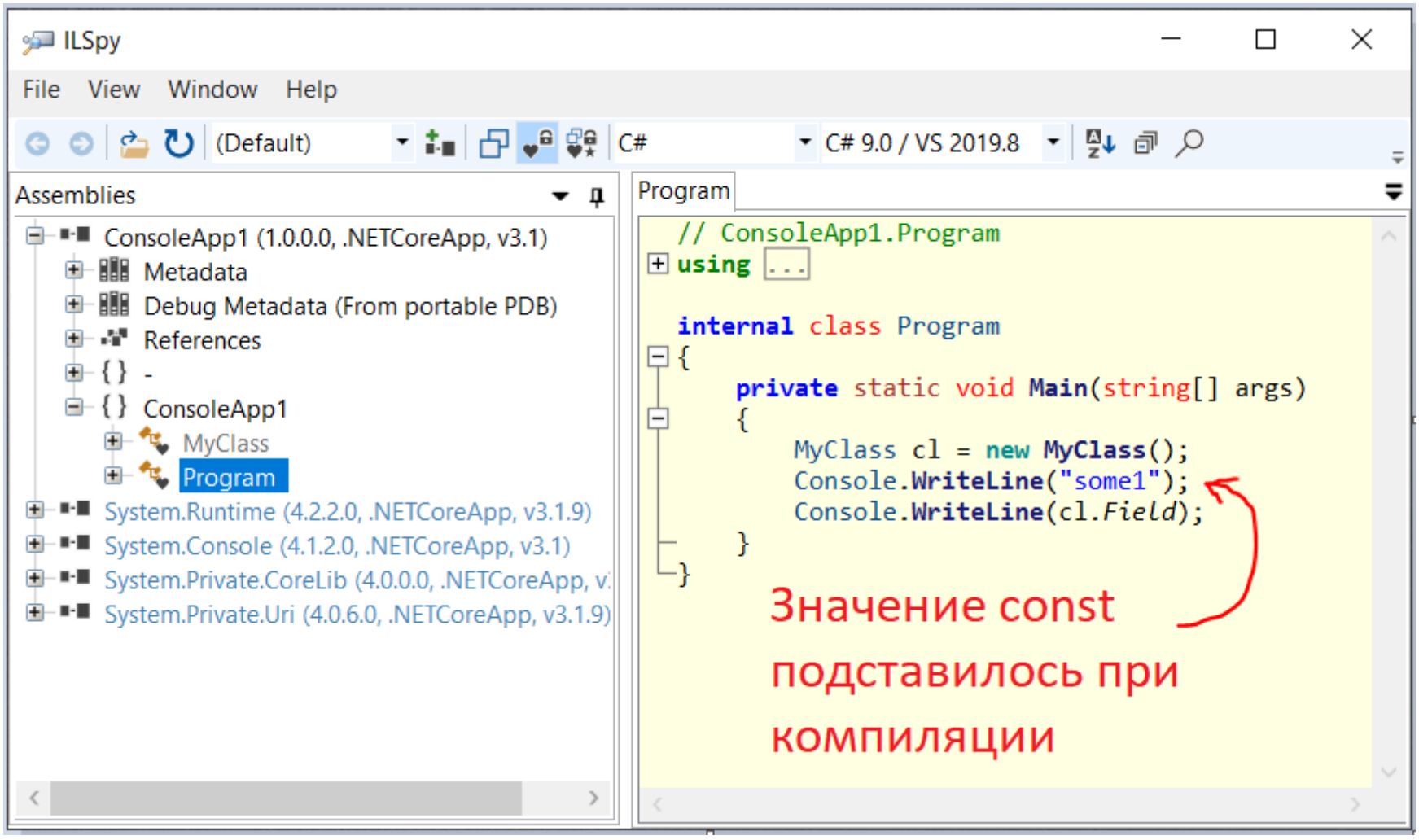
- хранятся в heap как refernce type
- при присвоении (передаче в метод) и сравнении ведут себя как value type

```
string a = "hello";
string b = "hello";

// string сравниваются как примитивный value type по значению
// (структуры с помощью == сравнивать нельзя)
Console.WriteLine(a == b);
// ==> true


var mc1 = new MyClass { Property = 1 };
var mc2 = new MyClass { Property = 2 };
// при сравнении reference type проверяется
// что переменные указывают на один и тот же объект в heap
Console.WriteLine(mc1 == mc2);
// ==> false
```

const vs readonly

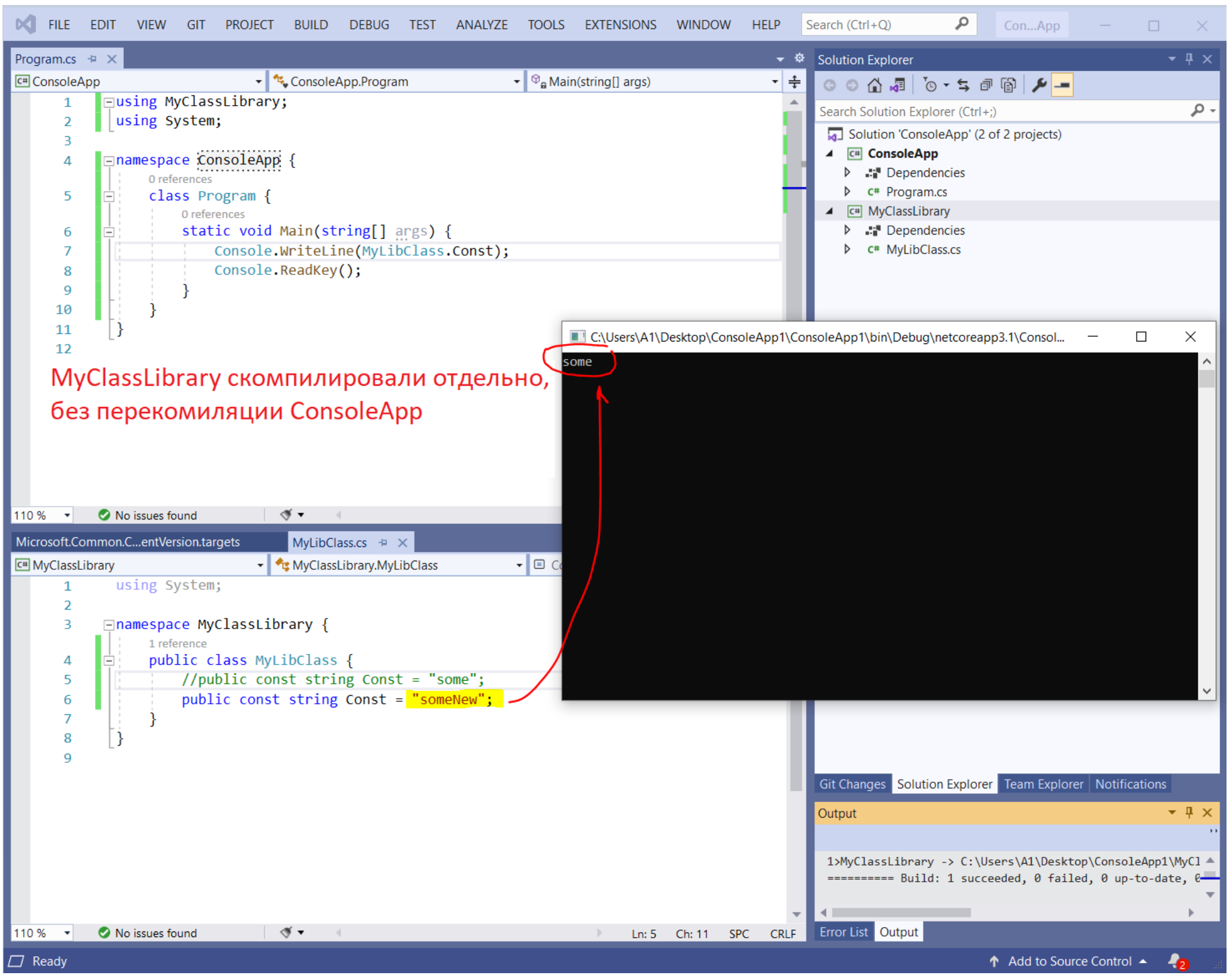
- const - значение подставляется при компиляции => установить можно только до компиляции
- readonly - установить значение можно только до компиляции или в конструкторе

```
class MyClass {
    public const string Const = "some1";
    public readonly string Field = "some2";
}

var cl = new MyClass();
Console.WriteLine(MyClass.Const);
Console.WriteLine(cl.Field);
```



Программа после компиляции. Компилятор подставил значение const.

Фокус-покус с подкладыванием dll библиотеки, без перекомпиляции основного проекта:



Значение const в библиотеке отличается от используемого в основном проекте.

ref и out

- ref и out позволяют внутри метода использовать new и для class и для struct
- out тоже что ref, только говорит о том что, метод обязательно пересоздаст переменную

```
struct MyStruc {
    public int Field;
}

class Program {
    static void Main(string[] args) {
        var ms = new MyStruc { Field = 1 };
        createNew(ms);
        Console.WriteLine(ms.Field);
        // ==> ms.Field = 1

        var ms2 = new MyStruc { Field = 1 };
        createNew2(ref ms2);
        Console.WriteLine(ms2.Field);
        // ==> ms2.Field = 2
    }

    static void createNew(MyStruc myStruc) {
        myStruc = new MyStruc { Field = 2 };
    }

    static void createNew2(ref MyStruc myStruc) {
        myStruc = new MyStruc { Field = 2 };
    }

    static void createNew3(out MyStruc myStruc) {
        // ошибка компиляции,
        // нужно обязательно использовать myStruc = new
    }
}
```

Ковариантность

Термин встречается только при обсуждении generic-ов.

```
interface IAnimal { }
class Cat : IAnimal {
    public void Meow() { }
}
class Dog : IAnimal {
    public void Woof() { }
}


// НЕ КОМПИЛИРУЕТСЯ, List - конвариантен
// не компилируется, потому что у List есть метод Add,
// который приводит к коллизиям (пример коллизии см. ниже)
List<IAnimal> animals = new List<Cat>();

// компилируется, IEnumerable - ковариантен
// у IEnumerable нет методов приводящих к коллизиям
IEnumerable<IAnimal> lst = new List<Cat>();
```

К каким коллизиям приводит метод Add в List:

```
// это компилируется и работает

List<Cat> cats = new List<Cat>();
cats.Add(new Cat());
List<Cat> animals = cats;
animals.Add(new Cat());

foreach (var cat in cats) {
    cat.Meow(); // в cats 2 кошки
}


// это НЕ КОМПИЛИРУЕТСЯ

List<Cat> cats = new List<Cat>();
cats.Add(new Cat());
List<IAnimal> animals = cats;
animals.Add(new Dog()); // это не порядок, потому что:

// перебираем
foreach (var cat in cats) {
    cat.Meow(); // в cats 1 кошка и 1 собака, у собаки нет метода Meow()
}
```

Публичные методы Object

- ToString
- GetType
- Equals
- GetHashCode

Про ToString и GetType спрашивать нечего.

Equals и GetHashCode нужны для сравнения объектов в коллекциях, linq, многопоточности. В основном переопределяют для ускорения, т.к. встроенные методы должны обеспечивать все многообразие платформы .Net. Разработчик может знать как быстрее посчитать hash для своих конкретных объектов.

Если объекты возвращают одинаковый GetHashCode не значит что они равны.

События, делегаты

```
class MyClass {
    public event Action<string> Evt;
    public void FireEvt() {
        if (Evt != null)
            Evt("hello");

        // Evt("hello") - на самом деле перебор обработчиков
        // можно сделать тоже самое вручную
        //foreach (var ev in Evt.GetInvocationList())
        //    ev.DynamicInvoke("hello");
    }

    public void ClearEvt() {
        // отписать всех подписчиков можно только внутри MyClass
        Evt = null;
    }
```

```
}


var cl = new MyClass();

// подписаться на событие
cl.Evt += (msg) => Console.WriteLine($"1 {msg}");
cl.Evt += (msg) => Console.WriteLine($"2 {msg}");

// подписаться и отписаться
Action<string> handler = (msg) => Console.WriteLine($"3 {msg}");
cl.Evt += handler;
cl.Evt -= handler;

cl.FireEvt();
// ==>
//  1 hello
//  2 hello


// это НЕ КОМПИЛИРУЕТСЯ
// на событие можно подписаться "+=" или описаться "-="
// отписать всех подписчиков можно только внутри MyClass
cl.Evt = null;
```

Finalizer (destructor) ~

- вызывается когда garbage collector доберется до объекта
- вызывается только автоматически средой .Net, нельзя вызвать самостоятельно
- нельзя определить для struct
- зачем может пригодиться переопределять finalizer: предпочтительней реализовать IDisposable. Встречаются идеи дублировать логику Dispose в finalizer, на случай если клиентский код не вызывал Dispose. Или вставлять в finalizer логирование времени жизни объекта для отладки.

trow vs "throw ex"

```
try {
    ...
} catch (Exception ex) {

    // это лучше, т.к. не обрезается CallStack
    throw;

    // обрезает CallStack
    throw ex;
}
```

Garbage collector

Коротко. heap большая, но все же имеет ограниченный размер, нужно удалять неиспользуемые объекты. Этим занимается Garbage collector. Деление объектов на поколения нужно для следующего:

- выявление есть ссылки на объект (объект используется) или его можно удалить - это трудозатратная задача
- поэтому имеет смысл делать это не для всех объектов в heap
- те объекты которые создали недавно (Generation 0) - вероятно это объекты используемые внутри метода, при выходе из метода они не нужны их можно удалить. Поэтому вначале искать объекты на удаление нужно в поколении Generation 0.

- те объекты которые пережили сборку мусора - называют объектами Generation 1.
- если Generation 0 почистили, а памяти не хватает. Приходится искать ненужные объекты среди тех которые пережили сборку - в Generation 1.
- если все равно нужно еще чистить, ищем среди тех кто пережили 2 сборки мусора - в Generation 2.

Порядок инициализации

- Delivered.Static.Fields
- Delivered.Static.Constructor
- Delivered.Instance.Fields
- Base.Static.Fields
- Base.Static.Constructor
- Base.Instance.Fields
- Base.Instance.Constructor
- Delivered.Instance.Constructor

```
class Parent {
    public Parent() {
        // нельзя вызывать virtual в конструкторе
        // конструктор базового класса вызывается
        // раньше конструктора наследника
        DoSomething();
    }
    protected virtual void DoSomething() {
    }
}

class Child : Parent {
    private string foo;
    public Child() {
        foo = "HELLO";
    }
    protected override void DoSomething() {
        Console.WriteLine(foo.ToLower()); //NullReferenceException
    }
}
```

Наследование в клиентском коде часто порождает излишнюю сложность (по мои наблюдениям, мое частное мнение), поэтому его нужно избегать. Для повторного использования кода лучше применить агрегацию (или, что хуже, композицию) - см. Наследование vs Композиция vs Агрегация.

ООП

- Абстракция - отделение идеи от реализации
- Полиморфизм - реализация идее разными способами
- Наследование - повторное использование кода лучше реализовать с помощью агрегации или, что хуже, композиции)
- Инкапсуляция - приватные методы

SOLID

- Single responsibility - объект, метод должны заниматься только одним своим делом, в противоположность антипатерну God-object

- Open closed principle - для добавления новых функций не должно требоваться изменять существующий код
- Liskov substitution - использовать базовый класс не зная о реализации наследника
- Dependency inversion principle - вначале интерфейсы, потом реализация, но не наоборот

Паттерны

Делятся на 3 типа

- порождающие (пример: фабрика)
- структурные (пример: декоратор)
- поведенческие (пример: цепочка обязанностей)

Что еще спрашивают

- IDisposable, try, catch, finally
- Напишите singleton (не забудьте про потокобезопасность и lock)
- домены приложений
- синхронизации потоков (mutex, semaphore и т.п.)
- Позитивные/негативные блокировки. Например: первый пользователь открыл форму на редактирование. Пока первый правит, второй успел внести изменения. Первый нажимает сохранить. Хорошо это или плохо? Какие варианты решения проблемы (если она есть)?
- SQL запросы, особенно с HAVING
- уровни изоляции БД

Литература

Stack and heap – .NET data structures

Boxing and Unboxing (C# Programming Guide)

Built-in reference types (C# reference)

Covariance and contravariance in generics

C# variance problem: Assigning List as List

Finalizers (C# Programming Guide)

Destructors in real world applications?

Virtual member call in a constructor

Наследование vs Композиция vs Агрегация

Fundamentals of garbage collection

Теги: .net, собеседование

Хабы: .NET, C#, Карьера в IT-индустрии