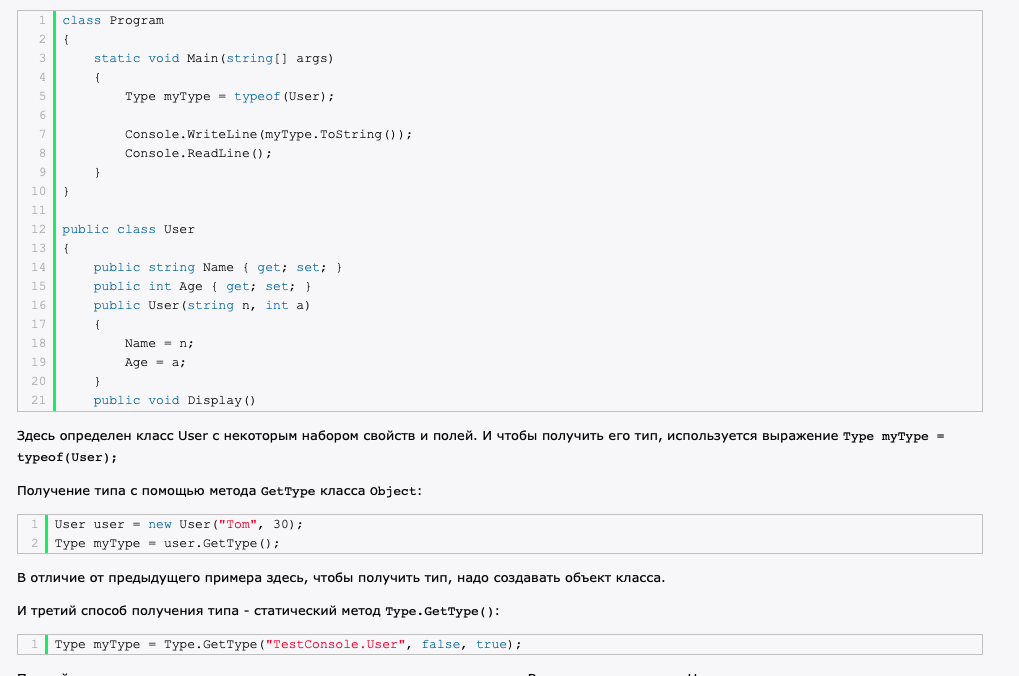
1. Что такое рефлексия

**Рефлексия** представляет собой процесс выявления типов во время выполнения приложения. Каждое приложение содержит набор используемых классов, интерфейсов, а также их методов, свойств и прочих кирпичиков, из которых складывается приложение. И рефлексия как раз и позволяет определить все эти составные элементы приложения.

1. Какие есть способы получения объекта типа Type. Приведите три

способа получения типа.

Чтобы управлять типом и получать всю информацию о нем, нам надо сперва получить данный тип. Это можно сделать тремя способами: с помощью ключевого слова **typeof**, с помощью метода **GetType()** класса Object и применяя статический метод Type.GetType().



1. Охарактеризуйте классы из пространства имен System.Reflection

Основной функционал рефлексии сосредоточен в пространстве имен **System.Reflection**. В нем мы можем выделить следующие основные классы:

* **Assembly**: класс, представляющий сборку и позволяющий манипулировать этой сборкой
* **AssemblyName**: класс, хранящий информацию о сборке
* **MemberInfo**: базовый абстрактный класс, определяющий общий функционал для классов EventInfo, FieldInfo, MethodInfo и PropertyInfo
* **EventInfo**: класс, хранящий информацию о событии
* **FieldInfo**: хранит информацию об определенном поле типа
* **MethodInfo**: хранит информацию об определенном методе
* **PropertyInfo**: хранит информацию о свойстве
* **ConstructorInfo**: класс, представляющий конструктор
* **Module**: класс, позволяющий получить доступ к определенному модулю внутри сборки
* **ParameterInfo**: класс, хранящий информацию о параметре метода

1. Как можно использовать класс System.Type?

Класс System.Type представляет изучаемый тип, инкапсулируя всю информацию о нем. С помощью его свойств и методов можно получить эту информацию. Некоторые из его свойств и методов:

* Метод **FindMembers()** возвращает массив объектов MemberInfo данного типа
* Метод **GetConstructors()** возвращает все конструкторы данного типа в виде набора объектов ConstructorInfo
* Метод **GetEvents()** возвращает все события данного типа в виде массива объектов EventInfo
* Метод **GetFields()** возвращает все поля данного типа в виде массива объектов FieldInfo
* Метод **GetInterfaces()** получает все реализуемые данным типом интерфейсы в виде массива объектов Type
* Метод **GetMembers()** возвращает все члены типа в виде массива объектов MemberInfo
* Метод **GetMethods()** получает все методы типа в виде массива объектов MethodInfo
* Метод **GetProperties()** получает все свойства в виде массива объектов PropertyInfo
* Свойство **IsAbstract** возвращает true, если тип является абстрактным
* Свойство **IsArray** возвращает true, если тип является массивом
* Свойство **IsClass** возвращает true, если тип представляет класс
* Свойство **IsEnum** возвращает true, если тип является перечислением
* Свойство **IsInterface** возвращает true, если тип представляет интерфейс

1. Как динамически загрузить сборку в приложение?

Для управления сборками в пространстве имен System.Reflection имеется класс **Assembly**. С его помощью можно загружать сборку, исследовать ее.

Чтобы динамически загрузить сборку в приложение, надо использовать статические методы **Assembly.LoadFrom()** или **Assembly.Load()**.

Метод LoadFrom() принимает в качестве параметра путь к сборке. Например, исследуем сборку на наличие в ней различных типов:

Метод Load() действует аналогично, только в качестве его параметра передается дружественное имя сборки, которое нередко совпадает с именем приложения: Assembly asm = Assembly.Load("TestConsole");

Получив все типы сборки с помощью метода GetTypes(), мы опять же можем применить к каждому типу все те методы, которые были рассмотрены в прошлой теме.

1. Что такое позднее(раннее) связывание?

**Позднее связывание** позволяет создавать экземпляры некоторого типа, а также использовать его во время выполнения приложения.

Использование позднего связывания менее безопасно в том плане, что при жестком кодировании всех типов (ранее связывание) на этапе компиляции мы можем отследить многие ошибки. В то же время позднее связывание позволяет создавать расширяемые приложения, когда дополнительный функционал программы неизвестен, и его могут разработать и подключить сторонние разработчики.

Ключевую роль в позднем связывании играет класс **System.Activator**. С помощью его статического метода **Activator.CreateInstance()**можно создавать экземпляры заданного типа.

