Темы урока

[**Разбор домашнего задания (реализация фабрики)**](#_6zj5vaog9t7n) **1**

[**Лямбда-выражения**](#_9a7uygyeofmc) **1**

[Самостоятельная работа](#_79zammv7tebd) 2

[**Библиотеки классов**](#_h472q9gq6owa) **2**

[.NET Standard vs .NET Core](#_c2o73u3v508g) 3

[Настройка зависимостей между проектами](#_xuwq2j2b6pk) 3

[Самостоятельная работа](#_wgocefnix6ez) 3

[Ссылки на DLL-библиотеки и NuGet-пакеты](#_lpohwq4t9y8s) 3

[**Chatbot: Постановка задачи**](#_3spk983r1trw) **3**

[Основная функциональность](#_754gz6ny2i0g) 3

[Дополнительные требования](#_h5d4qmrbggfo) 4

[**Домашнее задание**](#_l7mphfdcuudx) **4**

# Разбор домашнего задания (реализация фабрики)

Подробно рассмотреть реализацию фабрики.

ILogWriter лучше унаследовать от IDisposable, так как конкретная имплементация ILogWriter может использовать неуправляемые ресурсы. В нашем случае мы даже знаем, что так и будет! Ну а если неуправляемых ресурсов нет, всегда можно оставить тело метода Dispose пустым.

Абстрактный класс AbstractLogWriter остался как есть.  
Можно обратить внимание на унификацию формата через закрытое константное поле \_logRecordFormat.

Собственно по самой фабрике: здесь не получится сделать T GetLogWriter<T>, так как конструкторы разных имплементаций требуют наличия различных параметров (а писать вызывать конструктор new T() с параметрами уже нельзя). Поэтому наш метод возвращает интерфейс ILogWriter, который потом уже будет приводиться к запрошенному типу данных.

# Лямбда-выражения

Экземпляр делегата можно инициализировать лямбда-выражением.

Отличительной чертой лямбд является оператор **=>** , который делит выражение на **левую часть с параметрами** и **правую с телом метода**.

Например, если определён так класс-делегат

|  |
| --- |
| delegate int DoCalculation(int number1, int number2); |

то экземпляр может быть создан как обычным приравниванием метода с необходимой сигнатурой:

|  |
| --- |
| DoCalculation action1 = Sum; // assuming this method defined |

а может быть записан в более короткой форме лямбда-выражения:

|  |
| --- |
| DoCalculation action2 = (int x, int y) => x \* y; // method even isn’t required |

Если подходить строго, это именно лямбда-выражение.   
Но это может быть и лямбда-оператор, если мы заключим его в блочные скобки:

|  |
| --- |
| DoCalculation action3 =  (int x, int y) =>  {  int z = x \* y;  return z;  }; |

Допускается не указывать типы аргументов, ведь компилятор и так знает тип и сигнатуру вашего делегата.

В случае если имеется лишь один аргумент то можно опустить обрамляющие его скобки.

Если в сигнатуре делегата нет аргументов, то необходимо указать пустые скобки.

Код доступен **L15\_C11\_lambda\_expressions\_demo**.

## Самостоятельная работа

1. Переписать расчёт периметра и площади окружности на использование лямбда-выражений вместо методов класса.
2. Добавить функцию вычисление диаметра и также вывести результат расчёта диаметра на экран.

Решение: **L15\_C12\_lambda\_expressions\_SW**.

# Библиотеки классов

Библиотека классов определяет типы и методы, которые могут быть вызваны из любого приложения или других библиотек.

Если вы создадите библиотеку классов, вы сможете по своему усмотрению распространять ее как независимый компонент или включить в состав одного или нескольких приложений.

Чтобы создать библиотеку классов необходимо выбрать соответствующий тип проекта - Class Library.

Существует несколько возможностей создать библиотеку классов:

* Class Library (.NET Core)
* Class Library (.NET Standard)

## .NET Standard vs .NET Core

Рассказать про .NET Standard и его отличие от использования .NET Core.

## Настройка зависимостей между проектами

Чтобы подключить библиотеку классов к проекту, в котором планируется использовать её классы необходимо в Solution Explorer открыть контекстное меню на пункте Dependencies (зависимости) и выбрать пункт Add Reference (добавить ссылку).

В открывшемся диалоговом окне выбрать слева пункт Projects (проекты) и отметить галочками необходимые проекты в солюшене и нажать OK.

## Самостоятельная работа

За основу берётся код самостоятельной работы, выполненной последней на 15 ом уроке: классы Circle и CircleOperation.

* Класс Circle вынести в отдельную сборку с именем
* Calculator.Figure типа .NET Standard
* Класс CircleOperation вынести в отдельную сборку с именем Calculator.Operation типа .NET Core
* Создать консольное приложение .NET Core в которое поместить логику расчёта параметров окружности используя внешние классы.
* Добавить в соответствующие библиотеки классы Square и SquareOperation для описания квадрата и расчёта его периметра и площади.
* В консольном приложении также рассчитать и вывести параметры квадрата.

## Ссылки на DLL-библиотеки и NuGet-пакеты

Показать, что если сослаться на уже собранную библиотеку (DLL-файл), то по прежнему всё работает.

Показать как можно добавлять сторонние NuGet-пакеты в свой проект.

Можно в качестве примера добавить Newtonsoft.Json пакет и за одно рассказать про JSON-формат, сериализацию/десериализацию объектов (методы Convert.SerializeObject, Convert.DeserializeObject<T>)

# Chatbot: Постановка задачи

Разработать программу, которую можно было бы зарегистрировать в качестве бота одного из чат сервисов.

### Основная функциональность

* Принимать в сообщении будильник: сообщение и время срабатывания
* В назначенное время посылать в ответ сообщение-напоминание.

### Дополнительные требования

* Сообщения должны оставаться в хранилище программы даже после срабатывания.
* В первой версии приложения, хранилище будет in-memory коллекцией, однако оно должно быть написано так, чтобы обеспечить легкую замену другой реализацией в будущих версиях.

# Домашнее задание

1. Подумать, какие можно выделить интерфейсы и классы для решения задачи чат бота?
2. Как они будут взаимодействовать между собой?

На уроке будем обсуждать предложенные варианты, выработаем консолидированный подход.