



Лекция 4

Строки, дата и время в C#.

Паттерны GoF

# Строки

- Тип `System.String` — представляет неизменяемый упорядоченный набор символов. Является прямым потомком `Object` и ссылочным типом, по этой причине строки всегда размещаются в куче.
- Тип `String` реализует несколько интерфейсов: `IComparable /IComparable<String>`, `ICloneable`, `IConvertible`, `IEnumerable/IEnumerable<Char>` и `IEquatable<String>`.
- Может иметь значение `null`.
- Максимальный размер объекта `String` может составлять в памяти 2 Гб, или около 1 миллиарда символов.
- Создавать строки можно, как используя переменную типа `string` и присваивая ей значение, так и применяя один из конструкторов класса `String`:

```
var s1 = "hello";
```

```
var s2 = null;
```

```
var s3 = new String('a', 6); // результатом будет строка "aaaaaa"
```

```
var s4 = new String(new char[] { 'w', 'o', 'r', 'l', 'd' });
```

# Строка как набор символов

- Так как строка хранит коллекцию символов, в ней определен индекатор для доступа к этим символам:

```
public char this[int index] { get; }
```

- Применяя индекатор, мы можем обратиться к строке как к массиву символов и получить по индексу любой из ее символов:

```
var s1 = "hello";  
var ch1 = s1[1];  
Console.WriteLine(ch1);  
Console.WriteLine(s1.Length);
```



# Основные методы строк

- **Compare:** сравнивает две строки с учетом текущей локали пользователя
- **CompareOrdinal:** сравнивает две строки без учета локали
- **Contains:** определяет, содержится ли подстрока в строке
- **Concat:** соединяет строки
- **CopyTo:** копирует часть строки или всю строку в другую строку
- **Format:** форматирует строку
- **IndexOf:** находит индекс первого вхождения символа или подстроки в строке
- **Insert:** вставляет в строку подстроку
- **Join:** соединяет элементы массива строк
- **Replace:** замещает в строке символ или подстроку другим символом или подстрокой
- **Split:** разделяет одну строку на массив строк
- **Substring:** извлекает из строки подстроку, начиная с указанной позиции
- **ToLower:** переводит все символы строки в нижний регистр
- **ToUpper:** переводит все символы строки в верхний регистр
- **Trim:** удаляет начальные и конечные пробелы из строки

# Форматирование и интерполирование строк

## □ Форматирование

```
var output = String.Format("Имя: {0}  Возраст: {1}", person.Name,  
person.Age);
```

## □ Начиная с версии языка C# 6.0 была добавлена такая функциональность, как интерполяция строк

```
var result = $"Имя: {person.Name}  Возраст: {person.Age}";
```

```
var result = $"{x} + {y} = {x + y}"; // 8 + 7 = 15
```

```
var result = $"{person?.Name ?? "Имя по умолчанию"}";
```

```
var result = $"{number:+# ### ## ##}"; // +1 987 654 32 10
```



# StringBuilder

- Класс **StringBuilder**, представляет динамическую строку. При создании строки `StringBuilder` выделяет памяти больше, чем необходимо этой строке
- Методы:
  - **Insert**: вставляет подстроку в объект `StringBuilder`, начиная с определенного индекса
  - **Reove**: удаляет определенное количество символов, начиная с определенного индекса
  - **Replace**: заменяет все вхождения определенного символа или подстроки на другой символ или подстроку
  - **Append**: Добавляет единичный объект в массив символов, увеличивая его при необходимости.
  - **AppendFormat**: доабвляет подстроку в конец объекта `StringBuilder`
  - **ToString**: Версия без параметров возвращает объект `String`, представляющий массив символов объекта `StringBuilder`
  - **Clear**: Очищает содержимое объекта `StringBuilder`, аналогично назначению свойству `Length` значения 0
  - **Equals**: Возвращает `true`, только если объекты `StringBuilder` имеют одну и ту же максимальную емкость, емкость и одинаковые символы в массиве
  - **CopyTo**: Копирует подмножество символов `StringBuilder` в массив `Char`



□ Microsoft рекомендует использовать класс `String` в следующих случаях:

- При небольшом количестве операций и изменений над строками
- При выполнении фиксированного количества операций объединения. В этом случае компилятор может объединить все операции объединения в одну
- Когда надо выполнять масштабные операции поиска при построении строки, например `IndexOf` или `StartsWith`. Класс `StringBuilder` не имеет подобных методов.

□ Класс `StringBuilder` рекомендуется использовать в следующих случаях:

- При неизвестном количестве операций и изменений над строками во время выполнения программы
- Когда предполагается, что приложению придется сделать множество подобных операций



# Regex expressions (регулярные выражения)

- Регулярное выражение используется для проверки соответствия строки шаблону. Регулярное выражение — это последовательность символов, которая определяет шаблон. Шаблон может состоять из литералов, чисел, символов, операторов или конструкций. Шаблон используется для поиска соответствий в строке или файле. Регулярные выражения часто используются при проверке входных данных, анализе и поиске строк. Например, проверка достоверной даты рождения, номера социального страхования и так далее.

```
var s = "Бык тупогуб, тупогубенький бычок, у быка губа бела была тупа";  
var regex = new Regex(@"туп(\w*)");  
MatchCollection matches = regex.Matches(s);  
if (matches.Count > 0){  
    foreach (Match match in matches)  
        Console.WriteLine(match.Value);  
}
```

## Классы символов

[...]	Любой из символов, указанных в скобках	[a-z]	В исходной строке может быть любой символ английского алфавита в нижнем регистре
[^...]	Любой из символов, не указанных в скобках	[^0-9]	В исходной строке может быть любой символ кроме цифр
.	Любой символ, кроме перевода строки или другого разделителя Unicode-строки		
\w	Любой текстовый символ, не являющийся пробелом, символом табуляции и т.п.		
\W	Любой символ, не являющийся текстовым символом		
\s	Любой пробельный символ из набора Unicode		
\S	Любой непробельный символ из набора Unicode. Обратите внимание, что символы \w и \S - это не одно и то же		
\d	Любые ASCII-цифры. Эквивалентно [0-9]		
\D	Любой символ, отличный от ASCII-цифр. Эквивалентно [^0-9]		



## Символы повторения

$\{n,m\}$	Соответствует предшествующему шаблону, повторенному не менее $n$ и не более $m$ раз	
$\{n,\}$	Соответствует предшествующему шаблону, повторенному $n$ или более раз	
$\{n\}$	Соответствует в точности $n$ экземплярам предшествующего шаблона	
$?$	Соответствует нулю или одному экземпляру предшествующего шаблона; предшествующий шаблон является необязательным	Эквивалентно $\{0,1\}$
$+$	Соответствует одному или более экземплярам предшествующего шаблона	Эквивалентно $\{1,\}$
$*$	Соответствует нулю или более экземплярам предшествующего шаблона	Эквивалентно $\{0,\}$

## Символы регулярных выражений выбора

	Соответствует либо подвыражению слева, либо подвыражению справа (аналог логической операции ИЛИ).
(...)	Группировка. Группирует элементы в единое целое, которое может использоваться с символами *, +, ?,   и т.п. Также запоминает символы, соответствующие этой группе для использования в последующих ссылках.
(?...?)	Только группировка. Группирует элементы в единое целое, но не запоминает символы, соответствующие этой группе.

## Якорные символы регулярных выражений

^	Соответствует началу строкового выражения или началу строки при многострочном поиске.	^Hello	"Hello, world", но не "Ok, Hello world" т.к. в этой строке слово "Hello" находится не в начале
\$	Соответствует концу строкового выражения или концу строки при многострочном поиске.	Hello\$	"World, Hello"
\b	Соответствует границе слова, т.е. соответствует позиции между символом \w и символом \W или между символом \w и началом или концом строки.	\b(my)\b	В строке "Hello my world" выберет слово "my"
\B	Соответствует позиции, не являющейся границей слов.	\B(ld)\b	Соответствие найдется в слове "World", но не в слове "ld"



# RegexOptions



- **Compiled:** при установке этого значения регулярное выражение компилируется в сборку, что обеспечивает более быстрое выполнение
- **CultureInvariant:** при установке этого значения будут игнорироваться региональные различия
- **IgnoreCase:** при установке этого значения будет игнорироваться регистр
- **IgnorePatternWhitespace:** удаляет из строки пробелы и разрешает комментарии, начинающиеся со знака #
- **Multiline:** указывает, что текст надо рассматривать в многострочном режиме. При таком режиме символы "^" и "\$" совпадают, соответственно, с началом и концом любой строки, а не с началом и концом всего текста
- **RightToLeft:** приписывает читать строку справа налево
- **Singleline:** устанавливает однострочный режим, а весь текст рассматривается как одна строка



# Проверка валидности email

```
var pattern = @"^((\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*))"  
if (Regex.IsMatch(email, pattern, RegexOptions.IgnoreCase)){  
    Console.WriteLine("Email подтвержден");  
}  
else{  
    Console.WriteLine("Некорректный email");  
}
```

# Структура DateTime

```
var date1 = new DateTime(2015, 7, 20); // год - месяц - день
```

```
var date1 = new DateTime(2015, 7, 20, 18, 30, 25); // ... - час - минута - секунда
```

## □ Форматированный вывод

```
Console.WriteLine( dateTime.ToString( "dd.MM.yyyy" ) );
```

```
Console.WriteLine( dateTime.ToString( "HH:mm:ss" ) );
```

## □ Преобразование часовых поясов

```
var hwZone = TimeZoneInfo.FindSystemTimeZoneById("Hawaiian Standard Time");
```

```
Console.WriteLine("{0} {1} is {2} local time.",
```

```
    hwTime,
```

```
    hwZone.IsDaylightSavingTime(hwTime) ? hwZone.DaylightName :  
    hwZone.StandardName,
```

```
    TimeZoneInfo.ConvertTime(hwTime, hwZone, TimeZoneInfo.Local));
```



# Упаковка и распаковка значимых ТИПОВ ДАННЫХ

- Когда любой значимый тип присваивается к ссылочному типу данных, значение перемещается из области стека в кучу. Эта операция называется упаковкой (boxing)
- Когда любой ссылочный тип присваивается к значимому типу данных, значение перемещается из области кучи в стек. Это называется распаковкой (unboxing)



# Паттерны GoF (Gamma, Helm, Johnson, Vlissides)

## □ Преимущества использования паттернов:

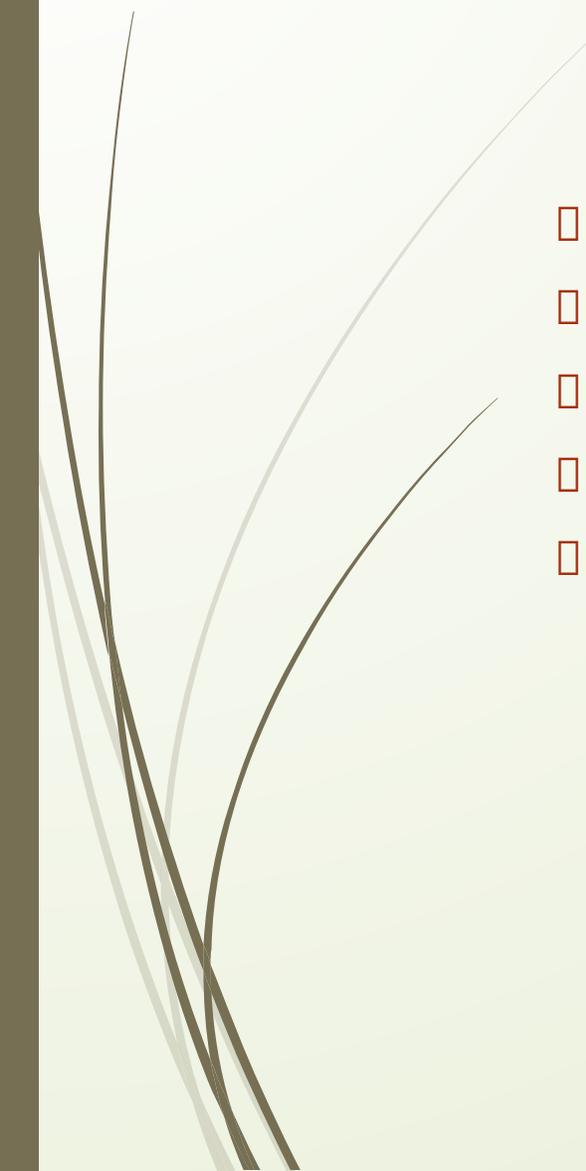
- **Проверенные решения.** Код более предсказуем когда программист использует готовые решения, вместо повторного изобретения велосипеда.
- **Стандартизация кода.** Вы делаете меньше ошибок, так как используете типовые унифицированные решения, в которых давно найдены все скрытые проблемы.
- **Общий язык.** Вы произносите название паттерна, вместо того, чтобы час объяснять другим членам команды какой подход вы придумали и какие классы для этого нужны.

## □ Недостаток:

- Обобщенное решение не панацея и в частных случаях бывает не достаточно эффективным



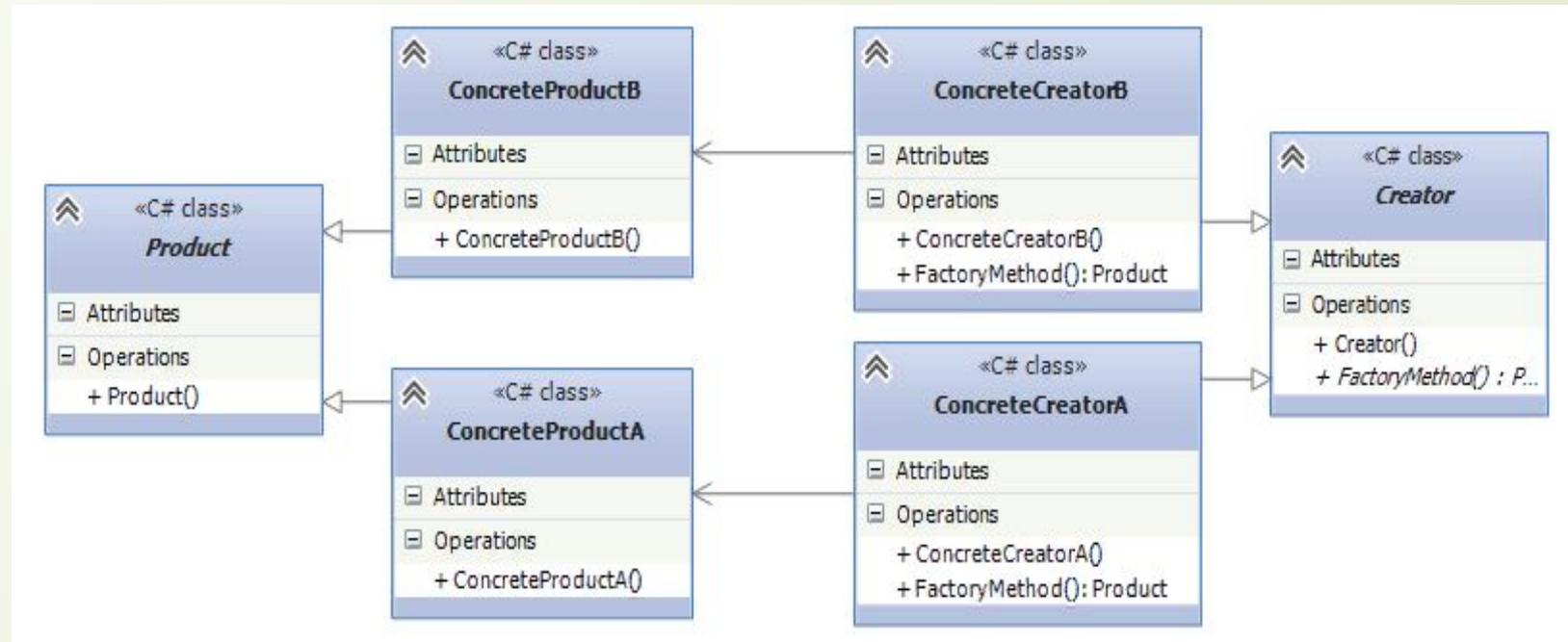
# Порождающие паттерны

- Абстрактная фабрика (Abstract Factory)
  - Строитель (Builder)
  - Фабричный метод (Factory Method)
  - Прототип (Prototype)
  - Одиночка (Singleton)
- 



# Factory Method

- Паттерн, определяющий интерфейс для создания объектов некоторого класса, но непосредственное решение о том, объект какого класса создавать происходит в подклассах. То есть паттерн предполагает, что базовый класс делегирует создание объектов классам-наследникам.
- **Применение**
  - Когда заранее неизвестно, объекты каких типов необходимо создавать
  - Когда система должна быть независимой от процесса создания новых объектов и расширяемой: в нее можно легко вводить новые классы, объекты которых система должна создавать.
  - Когда создание новых объектов необходимо делегировать из базового класса классам наследникам





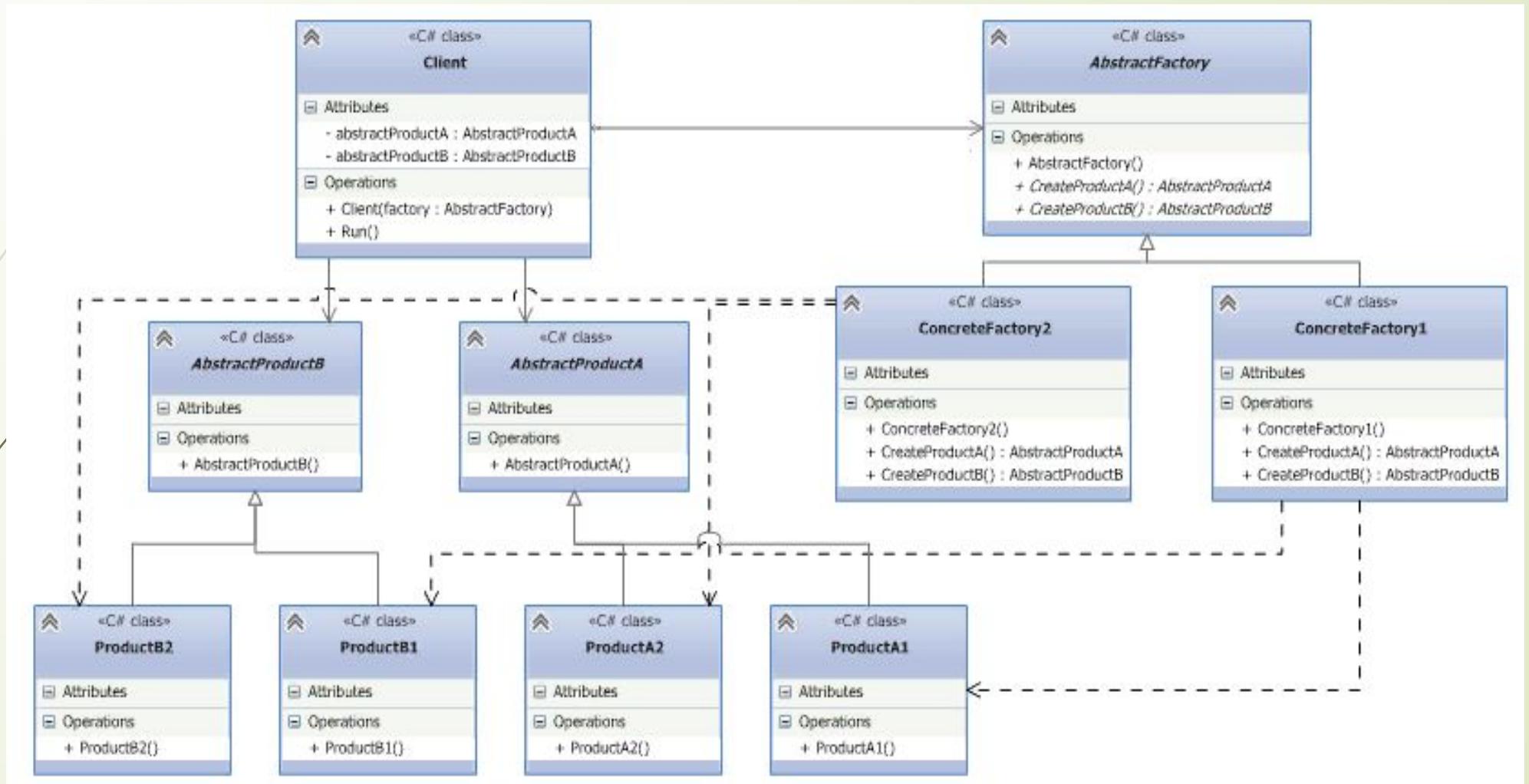
# Abstract Factory

Применение:

- Когда система не должна зависеть от способа создания и компоновки новых объектов
- Когда создаваемые объекты должны использоваться вместе и являются взаимосвязанными

Пример

- Проектирование игры, в которой пользователь должен управлять некими героями, при этом каждый герой имеет определенное оружие и определенную модель передвижения. Различные герои могут определяться комплексом признаков. Например, драгун может летать на драконе и управляться копьем, другой герой должен скакать на лошади и управлять мечом. Таким образом, получается, что сущность оружия и модель передвижения являются взаимосвязанными и используются в комплексе. То есть имеется один из доводов в пользу использования абстрактной фабрики. И кроме того, наша задача при проектировании игры абстрагировать создание героев от самого класса героя, чтобы создать более гибкую архитектуру.





# Singleton

- Одиночка (Singleton, Синглтон) - порождающий паттерн, который гарантирует, что для определенного класса будет создан только один объект, а также предоставит к этому объекту точку доступа.
- Синглтон позволяет создать объект только при его необходимости. Если объект не нужен, то он не будет создан. В этом отличие синглтона от глобальных переменных.



```
class Singleton
```

```
{
```

```
    private static Singleton instance;
```

```
    private Singleton()
```

```
    {}
```

```
    public static Singleton getInstance()
```

```
    {
```

```
        if (instance == null)
```

```
            instance = new Singleton();
```

```
        return instance;
```

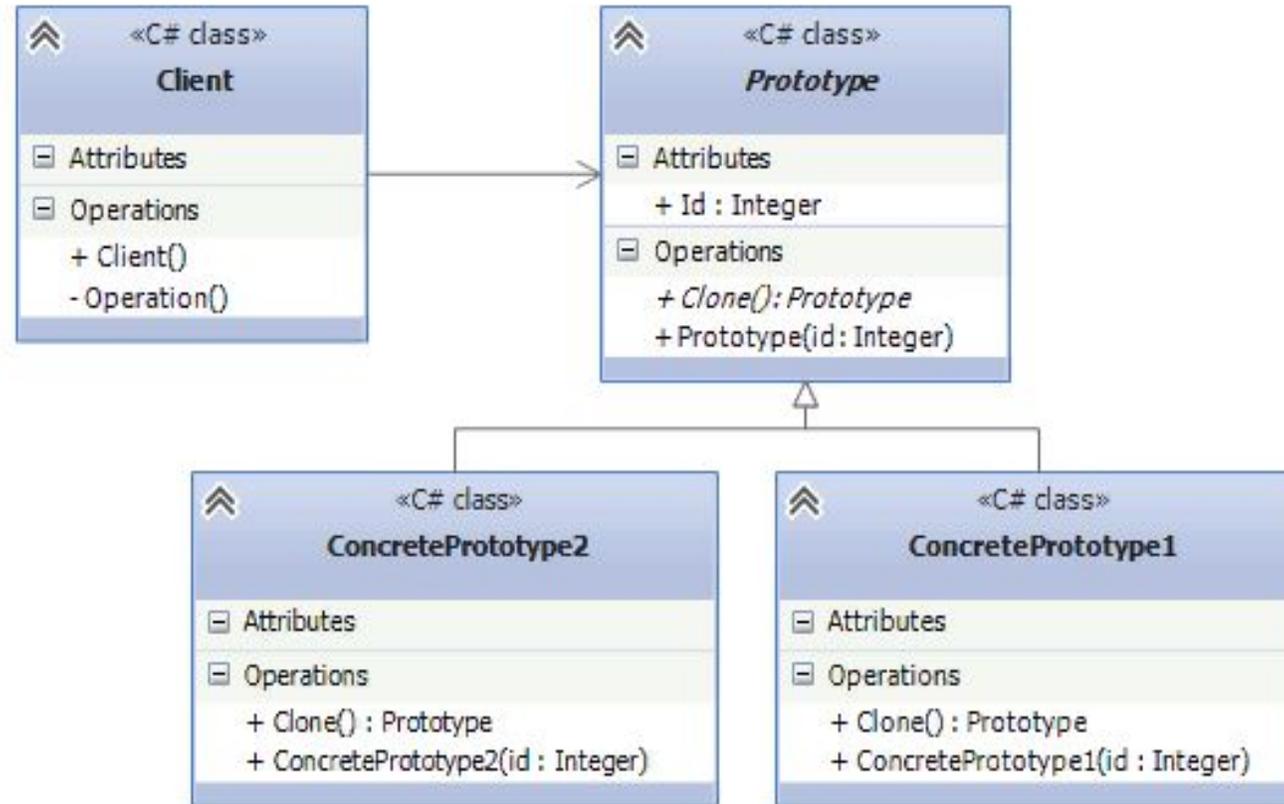
```
    }
```

```
}
```



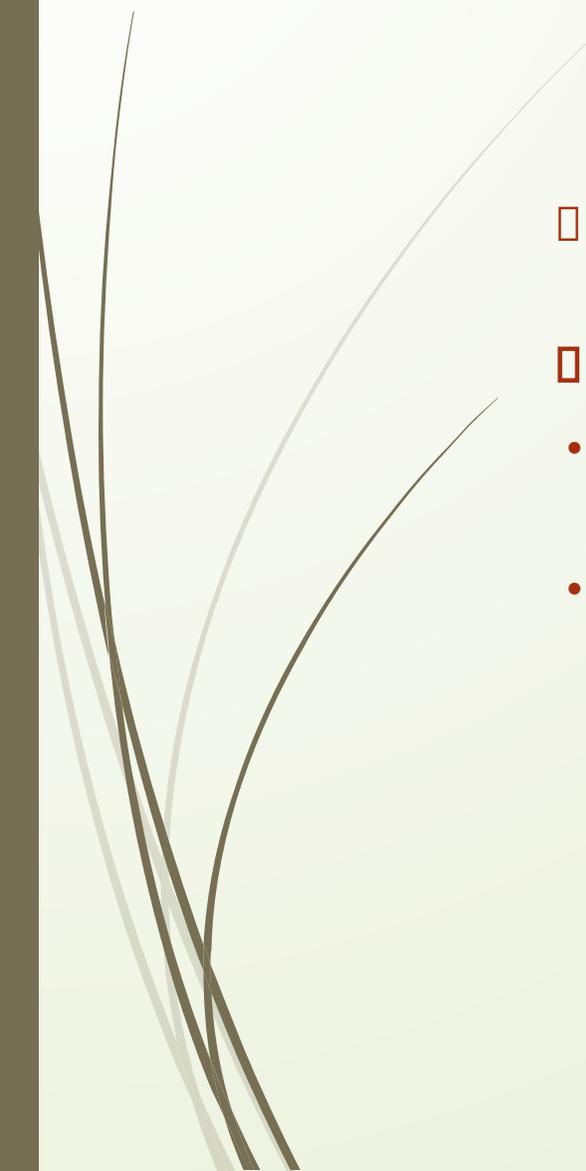
# Prototype

- ▣ **Прототип** — позволяет создавать новые объекты путем клонирования уже существующих. По сути данный паттерн предлагает технику клонирования объектов.
- ▣ **Участники**
  - Prototype: определяет интерфейс для клонирования самого себя, который, как правило, представляет метод Clone()
  - ConcretePrototype1 и ConcretePrototype2: конкретные реализации прототипа. Реализуют метод Clone()
  - Client: создает объекты прототипов с помощью метода Clone()

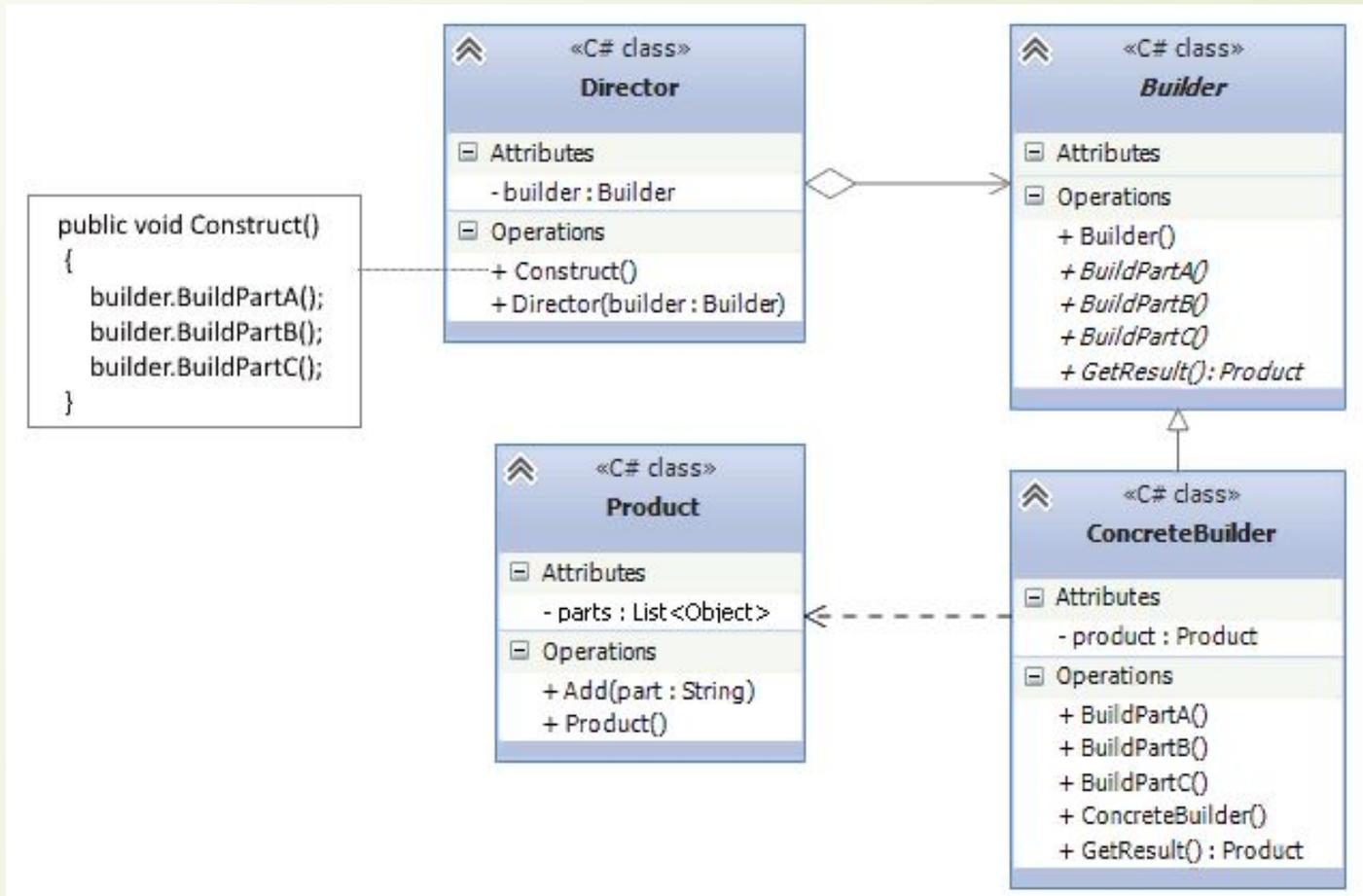




# Builder



- Шаблон проектирования, который инкапсулирует создание объекта и позволяет разделить его на различные этапы.
- **Применение:**
  - Когда процесс создания нового объекта не должен зависеть от того, из каких частей этот объект состоит и как эти части связаны между собой
  - Когда необходимо обеспечить получение различных вариаций объекта в процессе его создания





```
class Flour{}
class Salt{ }
class Additives{}
class Bread{
    public Flour Flour { get; set; }
    public Salt Salt { get; set; }
    public Additives Additives { get; set; }
}
abstract class BreadBuilder{
    public Bread Bread { get; private set; }
    public void CreateBread(){
        Bread = new Bread();
    }
    public abstract void SetFlour();
    public abstract void SetSalt();
    public abstract void SetAdditives();
}
```



```
class RyeBreadBuilder : BreadBuilder{
    public override void SetFlour(){
        this.Bread.Flour = new Flour { Sort = "Ржаная мука 1 сорт" };
    }

    public override void SetSalt(){
        this.Bread.Salt = new Salt();
    }
    public override void SetAdditives(){}
}

class Baker{
    public Bread Bake(BreadBuilder breadBuilder){
        breadBuilder.CreateBread();
        breadBuilder.SetFlour();
        breadBuilder.SetSalt();
        breadBuilder.SetAdditives();
        return breadBuilder.Bread;
    }
}
```