

Задание №4.

Наследование

Цели задания:

- Изучить механизм наследования объектно-ориентированного программирования.

Наследование — это свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью

Наследование выражает связь «общее-частное» и позволяет расширить функциональность базового класса в дочернем. Если реальные объекты предметной области не связаны как «общее-частное», то и в коде программы не надо использовать механизм наследования. Например, если объекты связаны как «часть-целое», в этом случае необходимо использовать агрегацию.

Модификатор доступа `protected` позволяет сделать доступными члены базового класса в дочерних классах. И только в дочерних.

Создание дочернего класса

1. Создать из ветки develop ветку Tasks/4_inheritance, перейти в новую ветку.
2. Добавьте в проект класс `PriorityOrder` (заказы приоритетного обслуживания). Приоритетный заказ также имеет поля `id`, времени создания, статуса, адреса доставки и списка товаров, но отличается от обычного заказа тем, что в приоритетном заказе пользователь может выбрать время доставки.
3. Чтобы не повторять реализацию тех полей и методов, которые совпадают с уже существующим классом `Order`, отнаследуйте класс `PriorityOrder` от класса `Order`. При необходимости, можете добавить в классы `Order` и `PriorityOrder` конструкторы без параметров.
4. Добавьте в класс `PriorityOrder` для поля с открытыми свойствами: желаемая дата доставки и желаемое время доставки. Дата доставки должна определяться стандартным типом данных `DateTime`, желаемое время доставки должно определять диапазоны:
 - 9:00 – 11:00
 - 11:00 – 13:00
 - 13:00 – 15:00
 - 15:00 – 17:00
 - 17:00 – 19:00
 - 19:00 – 21:00
5. Реализовать время доставки в качестве диапазона можно двумя способами – перечислением или строкой. В случае перечисления становится затруднительным именование элементов перечисления. В случае строки в дальнейшем вам придется обеспечить со стороны пользовательского интерфейса защиту, чтобы пользователь не мог задать значения кроме вышеперечисленных. Как видите, оба решения имеют свои минусы. Выбор варианта реализации остается за вами.
6. Добавьте в класс `PriorityOrder` конструктор по всем полям класса. Конструктор должен наследоваться от конструктора `Order` и инициализировать только те поля, которые объявлены в классе `PriorityOrder`.
7. Скомпилируйте программу. Проверьте правильность оформления кода и наличие всех комментариев. Сделайте коммит.

Работа с дочерними классами

1. Сначала мы попробуем работать с дочерними классами независимо от остального приложения. Создайте новый элемент управления `PriorityOrdersTab` и сверстайте его согласно макету:

Selected Order	Priority Options
ID: <input type="text"/>	Delivery Time: <input type="text" value="9:00 - 11:00"/>
Created: <input type="text"/>	
Status: <input type="text"/>	
Delivery Address	
Post Index: <input type="text"/>	
Country: <input type="text"/>	City: <input type="text"/>
Street: <input type="text"/>	
Building: <input type="text"/>	Apartment: <input type="text"/>
Order Items	
<div></div>	
Amount:	
4 999,90	
<input type="button" value="Add Item"/>	<input type="button" value="Remove Item"/>
<input type="button" value="Clear Order"/>	

2. Данный элемент предназначен исключительно для тренировки работы с дочерними классами, и в последующих заданиях будет удален. Элемент управления предназначен для работы только с одним объектом заказа `PriorityOrder`, поэтому добавьте в класс элемента поля типа `PriorityOrder`.

3. Обратите внимание, что элемент управления очень похож на элемент для отображения обычных заказов `Order`. Поэтому большую часть его реализации можно будет скопировать с вкладки `OrdersTab`. Однако, есть и отличия. Во-первых, это область `Priority Options` в верхнем левом углу, в которой указывается время доставки. Во-вторых, внизу добавлены кнопки для добавления и удаления товаров к заказу, а также кнопка очистки всех данных заказа `Clear Order`.

4. Реализуйте логику ввода и валидации данных согласно реализации класса `PriorityOrder`. Поля `Id` и `Created` должны быть доступными только для чтения, статус и время доставки должны выбираться из набора доступных вариантов и т.д.

5. Реализуйте логику кнопок добавления и удаления товаров Add Item и Remove Item. Кнопка добавления товара добавляет случайный новый товар в заказ. Для этого вы можете использовать товары из коллекции в классе Store (предварительно передав их на данную вкладку), либо создать и использовать генератор случайных товаров. Кнопка удаления удаляет выбранный в списке товар и меняет индекс выбранного элемента на следующий товар или, если следующего не существует, на последний товар списка.

6. Кнопка Clear Order должна не просто стирать данные из существующего заказа. В качестве практики, кнопка должна удалять старый объект заказа из поля вкладки, и создавать новый экземпляр заказа. С точки зрения реального приложения, нет причин удалять старый объект, и можно было бы просто сбросить в нём значения всех полей. Очистка данных существующего объекта или пересоздание объекта – равносильные варианты, при условии, что ссылки на существующий объект не хранятся в других частях приложения. Однако в рамках данного задания, реализуйте вариант пересоздания объекта заказа по нажатию кнопки.

7. Запустите программу и убедитесь, что новая вкладка работает правильно. Проверьте правильность верстки и оформления кода, структуру проекта. Сделайте коммит в репозиторий.

8. Если вы уверены в своих силах или если вам проще, вы можете реализовать вкладку Priority Orders аналогично вкладке Orders – вкладка может работать с множеством приоритетных заказов, все приоритетные заказы отображаются в DataGridView. Отличие реализации будет заключаться в том, что обычные заказы Orders создаются через вкладку CartsTab, а для приоритетных заказов надо будет добавить на вкладке Priority Orders отдельные кнопки для добавления и удаления новых заказов.

Работа с дочерними классами через ссылки на базовый класс

1. Важной особенностью наследования является то, что в переменных базового класса мы можем хранить объекты любых дочерних классов. Например, следующий код абсолютно корректен:

```
Order order1 = new Order();  
PriorityOrder priority2 = new PriorityOrder();  
order1 = new PriorityOrder();  
order1 = priority2;
```

Однако есть существенное ограничение – через переменную базового класса мы сможем обращаться только к тем полям дочернего объекта, которые унаследованы от базового класса. Другими словами, мы можем обратиться через переменную `order1` к свойствам `Id`, `Status` или списку товаров `Items`, но не сможем обратиться к времени доставки `DeliveryTime`. Обратиться к свойствам и методам дочернего класса можно только через переменные типа `PriorityOrder`.

Другое важное ограничение заключается в том, что хранение объектов дочернего типа в переменных базового класса не работает в обратную сторону. То есть, мы не можем хранить объекты базового класса в переменных дочернего:

```
PriorityOrder priority3 = new Order(); // Ошибка компиляции
```

Благодаря наследованию, дочерний класс является **расширением** функциональности базового класса, он является **разновидностью** базового класса. Поэтому нет ничего странного в том, чтобы хранить разновидность класса в переменных этого класса. Ведь всё, что есть в базовом классе, есть и в дочернем, а значит, обращение к любым полям и методам через переменную базового класса не может привести к ошибке. Однако если представить хранение объектов базового класса в переменных дочернего, то при попытке вызвать у базового объекта методы, которые появились только в дочернем классе, закономерно приведет к ошибке выполнения – исполняемая программа не может вызвать метод, которого просто нет в базовом классе. Поэтому в языке C# (и, как правило, других высокоуровневых языках) хранение базовых объектов в переменных дочернего класса запрещено синтаксически.

Третья важная особенность переменных базового класса заключается в том, что после помещения объекта дочернего класса в переменную базового, мы не можем простыми способами определить, объект какого именно класса (базового или дочернего) хранится в этой переменной. В Си++ это может стать проблемой для начинающего разработчика, но в C# у каждого объекта есть метод

GetType(), который возвращает тип данного объекта. Если мы захотим определить, какой объект хранится в переменной базового класса, мы можем написать следующий код:

```
var orderType = order1.GetType();
if (orderType == typeof(PriorityOrder))
{
    PriorityOrder priority = (PriorityOrder)order1;
    // Создали новую переменную с помощью явного преобразования
    // и теперь можем обращаться к любым методам дочернего класса
}
else
{
    // Иначе это обычный заказ, и тогда работаем
    // с объектом как с обычным заказом
}
```

В новых версиях C# можно использовать более лаконичную форму записи:

```
if (order is PriorityOrder priority)
{
    // Как и в примере выше, внутри if нам теперь доступна
    // переменная priority, через которую можем обращаться
    // к методам дочернего класса
}
else
{
    // Иначе это обычный заказ, и тогда работаем
    // с объектом как с обычным заказом
}
```

Всё вышеперечисленное позволяет нам:

- 1) Делать методы для работы с объектами базового класса, которые будут работать и с объектами дочерних классов. Например, если мы сделаем метод ShowOrderInfo(Order order), то этот метод может работать и с объектами PriorityOrder.
- 2) Хранить все объекты базовых и дочерних классов в общих коллекциях. Например, объекты классов Order и PriorityOrder могут храниться в общей коллекции List<Order>.

3) Делать общую обработку для таких общих коллекций. Например, мы можем перебрать в цикле все объекты коллекции `List<Order>` и просуммировать общую стоимость всех существующих заказов.

4) При необходимости забирать из общей коллекции объекты, определять их тип и преобразовывать к типу дочернего класса.

Как следствие, практически весь код, который мы ранее написали для класса `Order`, будет работать и с новым классом `PriorityOrder` – это еще одно важное преимущество наследования.

2. Добавим в основную программу понятие приоритетных покупателей – покупателей, чьи заказы являются приоритетными -, а также возможность создания приоритетных заказов.

3. Так как приоритетные заказы могут храниться в той же коллекции `List<Order>` `Orders` обычного покупателя, то создавать класс приоритетного покупателя `PriorityCustomer` смысла нет. Достаточно в класс `Customer` добавить логическое свойство `IsPriority`, которое будет указывать, является покупатель приоритетным в обслуживании или нет. По умолчанию свойство должно иметь значение `false`.

4. Добавьте на вкладку `CustomersTab` флажок `Is Priority`, который будет задавать покупателю значение соответствующего свойства:

Selected Customer

ID:

Full Name:

☒ Is Priority

Delivery Address

Post Index:

Country: City:

Street:

Building: Apartment:

5. На вкладке `CartsTab` измените логику создания заказов. Теперь, если пользователь нажмет кнопку `CreateOrder`, программа должна проверить, является ли текущий покупатель приоритетным. Если покупатель приоритетный, то программа должна создавать объект нового заказа `PriorityOrder` вместо обычного `Order`. После создания заказа и его инициализации, заказ должен помещаться в список заказов покупателя.

6. Запустите программу и убедитесь, что программа работает корректно. Главное, убедитесь, что даже при создании заказов типа `PriorityOrder`, все новые

заказы отображаются на вкладке Orders. Несмотря на создание нового типа заказов, ранее написанная логика по отображению и редактированию заказов должна работать как и прежде. Фактически, механизм наследования позволил нам не дублировать логику целой вкладки OrdersTab для нового типа заказов. Однако, определенные изменения во вкладку придется внести.

7. Вкладка OrdersTab работает с любыми заказами как с объектами класса Order. Мы можем редактировать любые данные PriorityOrder, которые унаследованы от базового класса. Но пока мы не можем редактировать те данные, которые являются уникальными для PriorityOrder.

8. Добавьте в верстку OrdersTab выпадающий список с выбором времени доставки по аналогии с ранее сделанной вкладкой PriorityOrdersTab:

Selected Order		Priority Options	
ID:	<input type="text" value="10269"/>	Delivery Time:	<input type="text" value="11:00 - 13:00"/>
Created:	<input type="text" value="2022.10.12 10:12:05"/>		
Status:	<input type="text" value="New"/>		
Delivery Address			
Post Index:	<input type="text"/>		
Country:	<input type="text"/>	City:	<input type="text"/>
Street:	<input type="text"/>		
Building:	<input type="text"/>	Apartment:	<input type="text"/>

9. Так как в зависимости от типа заказа, мы будем управлять видимостью новых элементов управления, рекомендуется три новых элемента разместить на элементе Panel. В таком случае, чтобы отключить видимость элементов нам достаточно будет поменять значение свойства Visible для одной панели, а не у каждого из трёх элементов.

10. Измените логику обработки смены выбранного заказа на вкладке. Теперь, если пользователь выберет заказ в таблице заказов, программа должна определить, является ли выбранный заказ приоритетным (механизм GetType(), описанный выше, или ключевые слова is и as). Если заказ приоритетный, то объект приоритетного заказа должен сохраниться в уже существующем поле _selectedOrder и в поле типа PriorityOrder _selectedPriorityOrder, которое надо будет добавить. То есть, один и тот же объект заказа будет одновременно храниться в двух полях, но в одном поле мы сможем работать с объектом заказа как с обычным заказом, а в тех случаях, когда нам понадобится обратиться к данным PriorityOrder, мы будем работать с объектом через второе поле.

11. Если выбранный заказ не приоритетный, то поле _selectedPriorityOrder должно сбрасываться в null. В зависимости от того, является новый выбранный заказ приоритетным или нет, должна меняться видимость выпадающего

списка Delivery Time и его подписей. Если заказ приоритетный, то выпадающий список Delivery Time должен быть видимым пользователю, и также проинициализированным значением из заказа.

12. Есть несколько способов реализовать описанную выше логику: написать алгоритм непосредственно в обработчике DataGridView_SelectedIndexChanged, либо реализовать определение типа заказа внутри сеттера свойства Order SelectedOrder вкладки OrdersTab. Кроме того, логику можно разделить на несколько последовательных методов, а вместо поля _selectedPriorityOrder (то есть постоянного хранения объекта внутри вкладки) внутри методов можно использовать логику определения типа заказа _selectedOrder и делать преобразование (то есть, хранить заказ по общей ссылке и преобразовывать только при необходимости). Все перечисленные реализации верные, но автор предпочел бы реализацию на основе сеттера свойства и выделения нового поля под приоритетный заказ.

13. Добавьте логику выбора времени доставки. Если пользователь выбирает другое время доставки, новое время должно сохраниться в приоритетном заказе.

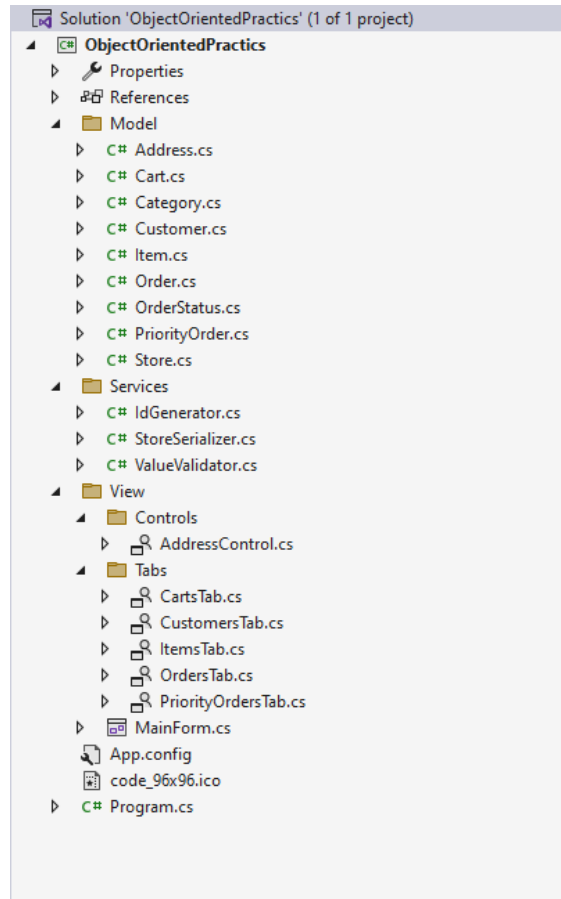
14. Вкладку PriorityOrdersTab теперь можно удалить из главного меню, но сам класс вкладки пока оставить в программе – для защиты задания перед преподавателем.

15. Таким образом, мы расширили функциональность нашей программы новым типом заказов. Без наследования нам бы пришлось дублировать реализацию практически всей программы для того, чтобы пользователь работал с приоритетными заказами. Однако благодаря наследованию, нам было достаточно добавить только ту логику, которая работает с новой функциональностью дочернего класса – а работа с унаследованными полями обеспечивается ранее написанным кодом.

16. Запустите программу и проверьте правильность работы. Если в предыдущем задании вы реализовывали поиск заказов, убедитесь, что он корректно работает в новой версии. Проверьте правильность верстки и её адаптивность. Проверьте структуру проекта, правильность оформления кода и наличие всех комментариев. Сделайте коммит.

Проектная документация

1. В результате выполнения всех заданий, структура проекта должна быть примерно следующей:



2. По результатам всех заданий нарисуйте новую диаграмму классов.
3. На ней должны быть отражены новые классы, а также связи между ними.
4. Обозначение наследования на диаграммах классов см. в книге Буча «Язык UML. Руководство пользователя».
5. Обратите внимание, что новые поля и свойства добавились в ранее написанные классы – изменения должны быть показаны на диаграмме.
6. Диаграмму сохраните под названием «Practics4.*» в папке doc. Сохраните диаграмму как в формате программы, которую вы будете использовать, так и в

формате png или jpeg, чтобы диаграмму можно было просмотреть онлайн через GitHub.

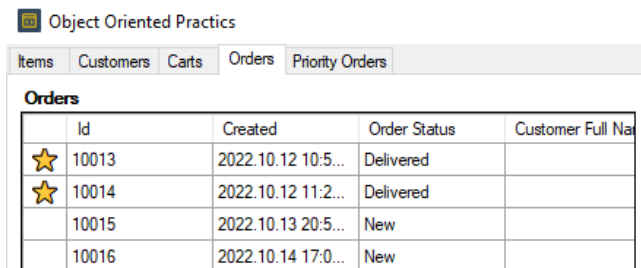
7. Выполнить сливание текущей ветки с веткой develop.

Дополнительные задания

1. Реализуйте или исправьте ранее реализованный механизм сохранения и загрузки пользовательских данных. Если вы реализовывали сохранение и загрузку данных без использования сторонних библиотек, задача сохранения и загрузки списка `List<Order>`, в котором одновременно хранятся объекты `Order` и `PriorityOrder`, может оказаться нетривиальной. Кроме того, не все сторонние библиотеки умеют корректно сохранять наследуемые объекты. Библиотека `Newtonsoft.JSON.NET` умеет правильно сохранять и загружать подобные коллекции, однако может потребоваться дополнительная настройка сериализатора – установить свойство `ValueTypeHandling` в значение `All`.

2. Пользователем нашей системы является не покупатель, а оператор (магазина или сервиса доставки). При обработке заказов оператор должен в первую очередь обрабатывать приоритетные заказы – а для этого он должен их видеть в таблице заказов. Реализуйте:

- Флажок `Show Only Priority Orders` под таблицей. Флажок включает или выключает отображение только приоритетных заказов. Важно: при включении флажка, заказы в таблице будут идти в другом порядке и количестве, чем они идут в общем списке всех заказов. А, значит, вы должны обеспечить правильную работу события `DataGridView_SelectedIndexChanged`, так как индексы выбранного в таблице заказа и заказа в общем списке теперь совпадать не будут.
- Новый столбец в начале таблицы, в котором будет показываться звездочка, если этот заказ является приоритетным:



	Id	Created	Order Status	Customer Full Name
★	10013	2022.10.12 10:5...	Delivered	
★	10014	2022.10.12 11:2...	Delivered	
	10015	2022.10.13 20:5...	New	
	10016	2022.10.14 17:0...	New	