**Introduction**

MSDN describes [ObservableCollection](http://msdn.microsoft.com/en-us/library/ms668604.aspx) as a dynamic data collection which provides notifications when items get added, removed, or when the whole list is refreshed.

[ObservableCollection](http://msdn.microsoft.com/en-us/library/ms668604.aspx) is fully bindable. It implements both [INotifyPropertyChanged](http://msdn.microsoft.com/en-us/library/system.componentmodel.inotifypropertychanged.aspx) and [INotifyCollectionChanged](http://msdn.microsoft.com/en-us/library/system.collections.specialized.inotifycollectionchanged.aspx), so whenever the collection is changed, appropriate notification events are fired off immediately and bound objects are notified and updated.

This scenario works in most cases but sometimes it would be beneficial to postpone notifications until later or temporarily disable them all together. For example, until a batch update is finished. This notification delay could increase performance as well as eliminate screen flicker of updated visuals. Unfortunately, the default implementation of [ObservableCollection](http://msdn.microsoft.com/en-us/library/ms668604.aspx) does not provide this functionality.

ObservableCollectionEx is designed to provide this missing functionality. ObservableCollectionEx is designed as a direct replacement for [ObservableCollection](http://msdn.microsoft.com/en-us/library/ms668604.aspx), is completely code compatible with it, and also provides a way to delay or disable notifications.

**Background**

In order to postpone notifications, we have to temporarily reroute them to a holding place and fire them all once delay is no longer required. At the same time, we need to continue to provide normal behavior and notifications for other consumers of the collection which do not require delay.

This could be achieved if we have multiple objects acting like a shell and manipulating the same collection. One instance will contain the element’s container and be a host for all of the notification events which consumers will be subscribed to, and other instances of the shell will handle disabled and delayed events. These extra shells reference the same container but instead of firing events which consumer handlers attached to, they will call its own handlers which either collect these events or discard them.

The ObservableCollection implementation is based on a Collection which implements functionality, and ObservableCollection implements notifications. The Collection class is implemented as a shell around the IList interface. It contains a reference to a container which exposes the IList interface and manipulates this container through it. One of the constructors of the Collection class takes List as a parameter and allows this list to be a container for that Collection. This creates a way to have multiple Collection instances to manipulate the same container, which perfectly serves our purpose.

Unfortunately, this ability is lost in the ObservableCollection implementation. Instead of assigning IList to be a container for the instance, it creates a copy of that List and uses that copy to store elements. This limitation prevents us from inheriting from the ObservableCollection class.

ObservableCollectionEx is based on a Collection class as well, and implements exactly the same methods and properties that ObservableCollection does.

In addition to these members, ObservableCollectionEx exposes two methods to create disabled or delayed notification shells around the container. Methods of the shell created by DisableNotifications() produce no notifications on either INotifyPropertyChanged or INotifyCollectionChanged.

Calls to the methods of the shell created by DelayNorifications() produce no notifications until this instance goes out of scope or IDisposable.Dispose() has been called on it.

**How it works**

Except for a few performance tricks, ObservableCollectionEx behaves exactly as the ObservableCollection class. It uses Collection to perform its operations, notifies consumers via INotifyPropertyChanged and INotifyCollectionChanged, and creates a copy of the List if you pass it to a constructor.

The differences starts when the DelayNotifications() or DisableNotifications() methods are called. This method creates a new instance of the ObservableCollectionEx object and passes its constructor a reference to the original ObservableCollectionEx object, and the Boolean parameter that specifies if notifications are disabled or postponed. This new instance will have the same interface as the original, the same element’s container, but none of the consumer handlers attached to the CollectionChanged event. So when methods of this instance are called and events are fired, none of these are going anywhere but to temporary storage.

Once updates are done, and either this instance goes out of scope or Dispose() has been called, all of the collected events are combined into one and fired on CollectionChanged and PropertyChanged of the original object notifying all of the consumers about changes.

**Using the code**

The easiest way to include this class into your project is by installing the [Nuget](http://www.nuget.org/) package available at this [link](http://www.nuget.org/List/Packages/ObservableCollectionEx).

ObservableCollectionEx should be used exactly as ObservableCollection. It could be instantiated and used in place of ObservableCollection, or it could be derived from it. No special treatment is required.

In order to postpone notifications, it is recommended to use the using() directive:

http://www.codeproject.com/images/minus.gifCollapse | [Copy Code](http://www.codeproject.com/KB/collections/ObservableCollectionEx.aspx)

ObservableCollectionEx<T> target = new ObservableCollectionEx<T>();

using (ObservableCollectionEx<T> iDelayed = target.DelayNotifications())

{

iDelayed.Add(item0);

iDelayed.Add(item0);

iDelayed.Add(item0);

}

Due to the design of notification arguments, it is not possible to combine different operations together. For example, it is not possible to Add and Remove elements on the same delayed instance unless Dispose() has been called in between these calls. Calling Dispose() will fire previously collected events and reinitialize operation.

http://www.codeproject.com/images/minus.gifCollapse | [Copy Code](http://www.codeproject.com/KB/collections/ObservableCollectionEx.aspx)

ObservableCollectionEx<T> target = new ObservableCollectionEx<T>();

using (ObservableCollectionEx<T> iDelayed = target.DelayNotifications())

{

iDelayed.Add(item0);

iDelayed.Add(item0);

}

using (ObservableCollectionEx<T> iDelayed = target.DelayNotifications())

{

iDelayed.Remove(item0);

iDelayed.Remove(item0);

}

using (ObservableCollectionEx<T> iDelayed = target.DelayNotifications())

{

iDelayed.Add(item0);

iDelayed.Add(item0);

iDelayed.Dispose();

iDelayed.Remove(item0);

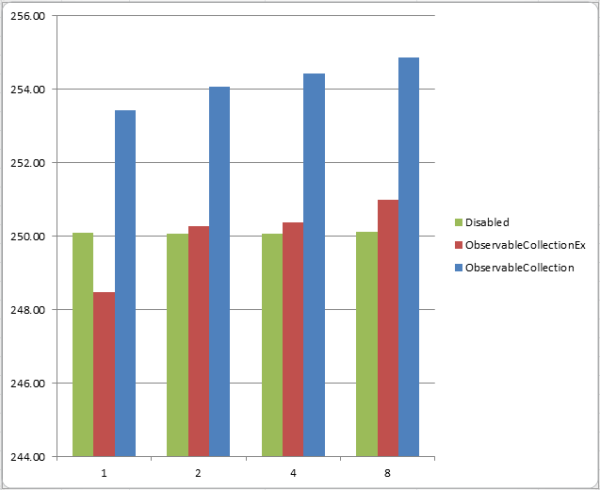
iDelayed.Remove(item0);

}

**Performance**

In general, both ObservableCollection and ObservableCollectionEx provide comparable performance. Testing has been done using an array of 10,000 unique objects. Both ObservableCollection and ObservableCollectionEx where initialized with this array to pre allocate storage so it is not affecting timing results. Application has been run about dozen times to let JIT to optimize the executable before the test results were collected.

The test consisted of 10,000 Add, Replace, and Remove operations. Timing has been collected using the Stopwatch class and presented in milliseconds.



The value on the left represents the number of milliseconds it took to complete the test (Add, Replace, and Remove). The value on the bottom specifies the number of notification subscribers (handlers added to the CollectionChanged event).

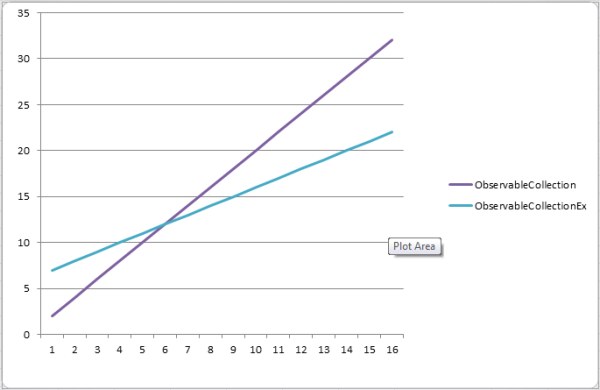
As you can see from the graph, the performance of the interface with disabled notifications does not depend on the subscribers. Due to several performance enhancements, ObservableCollectionEx performs slightly better than ObservableCollection regardless of the number of subscribers but it obviously loses the Disabled interface once there is more than one subscriber.

The performance of ObservableCollectionEx when notifications are delayed is different compared to the results described above. Since notification is called only once, it saves some time but it requires some extra processing to unwind saved notifications. Time spent on notifications for ObservableCollection and ObservableCollectionEx are described by the following equitation:

**ObservableCollection**: overhead = (**n** \* **a**) + (**n** \* **b**)

**ObservableCollectionEx**: overhead = **a** + **c** + (**n** \* **b**)

Where **a** is a constant overhead required to execute notification, **n** is number of changed elements, **b** is the cost of redrawing each individual element, and **c** the overhead required to execute delayed notification.



The value on the left represents the time required to complete notifications. The value on the bottom specifies the number of changed elements.

In these equations, values **a** and **c** are constants so the performance depends only on two elements: **b** – the time required to redraw each individual element, and **n** – the number of notified elements. As you know from calculus, **b** controls how steep the raise of the graph is. So when the time required to redraw each element (b) increases, these two lines meet sooner. It means it takes less changed elements to see performance benefits.