# Title

* Sorry guys, I thought I’d try making the slides immutable, so I had to copy it every time I changed it.
* Since we’ve had a lot of recent presentations about functional programming, Scala, akka, etc. I wanted to present on one of the basic tenants of functional programming because I think that will help us better understand some of the functional concepts we’ve seen in those presentations. So today I want to discuss Immutability, but not just the kind of immutability we’re already familiar with. First, let’s quickly review what immutability is. (NEXT SLIDE)

# Some Review

- I think you guys all know what immutability is. We’ve all been exposed to it from using Strings and dates. Here are just a few examples of some of the immutable Types we routinely use (NEXT SLIDE)

# Examples

* Some of these you probably knew were immutable. I found it interesting to learn that Tuples and anonymous objects are also immutable. Can anyone name some other immutable types that we often use?
* But we’ve already learned about immutable objects, so today I’m going to cover something you might not be as familiar with: immutable collections. (NEXT SLIDE)

# Immutable Collections

* Can anyone tell me what an immutable collection is? Let’s just put the two words together, it must mean it’s a collection that can’t change.
* When I first heard the term “immutable collection” it didn’t make any sense to me. I thought “what use is a collection if you can’t change it?” But think about how you’d use it the same way you use strings or DateTimes: the collection can still be added to and removed from, but doing so will create a new instance. The original collection that you reference doesn’t change, and instead you’ll get back a reference to a new collection containing your changes.
* Can anyone tell me some examples of where immutable collections would be useful? (When you need a “snapshot” i.e. places that clone the collection with a ToList() call, when you are using the collection from multiple threads).

# Demo

* Let’s try writing an immutable collection ourselves to get an idea of how it might work. When I first thought about how an immutable collection might be implemented I thought about copying a list every time it needs to change. (DEMO SlowImmutableList) This is a really inefficient way to get immutability, and if we were stuck with this we’d quickly run into performance issues. Imagine adding a lot of elements to this list, and having to clone it for every single add. (Ask questions about the Big O of adding, removing, getting the count, and looking up the element at a specific location).
* Let’s take a look at the traditional functional list that you’ll find in a lot of functional languages. (DEMO FunctionalList) You can see this is much better than our SlowImmutableList. Adding is just as fast (or faster) than adding to a regular List, and we aren’t creating a bunch of garbage in memory every time. In fact, this list is very memory efficient because the items are shared. This kind of collection is also easy to use recursively, which is another aspect of functional programming. (Ask questions about the Big O of adding, removing, getting the count, and looking up the element at a specific location).

# Immutable Trees

So the functional list is one way of creating an immutable list. Another common way of creating immutable data structures is to use a binary tree. To keep the collection immutable, the nodes that change need to be cloned when an item is added or removed, but the rest of the tree stays the same. This allows most of the collection to be shared, which produces much less garbage than cloning the full collection.

In the example shown, we have a sorted binary tree that contains several integers. How would this tree change if we wanted to add the number 11 to this collection? (NEXT SLIDE)

# Spine Rewrite

Because we have to modify the 10 node to attach the 11, we create a new one, which means the 13 node needs to be updated, and the 8 node at the root is updated. This is called a Spine Rewrite, because one path through the tree is modified while the rest remains unchanged.

Instead of writing a custom immutable collection using a binary tree, I’m going to show you a nuget package that contains immutable collections that follow this approach. (NEXT SLIDE)

# Microsoft.Bcl.Immutable

This nuget package adds the System.Collections.Immutable namespace to complement the existing System.Collections.Concurrent data structures. All of these data structures are implemented by using an AVL tree, which is a form of binary tree (basically a rigidly balanced binary tree). Wikipedia has a detailed explanation of AVL trees if you want to know more.

# Performance

- As we discussed with the naïve example and the FunctionalList, there are performance tradeoffs when using immutable collections, and the same applies to the immutable collections library. For example, this table shows the performance of adding to each type of collection. This is something to consider when deciding what type of collection to use.

- Now for an example of the immutable collections library in action, let’s try creating an EventAggregator that uses an ImmutableList.