Interfaces and generics

Introducing Interfaces

- Interfaces define a set of characteristics and behaviors
  - Member signatures only
  - No implementation details
  - Cannot be instantiated
- Interfaces are implemented by classes or structs
  - Implementing class or struct must implement every member
  - Implementation details do not matter to consumers
  - Member signatures must match definitions in interface
- By implementing an interface, a class or struct guarantees that it will provide certain functionality

Defining Interfaces

- Use the **interface** keyword

```
public interface IBeverage
{
    // Methods, properties, events, and indexers.
}
```

- Specify an access modifier:
  - public
  - internal
- Add interface members:
  - Methods, properties, events, and indexers
  - Signatures only, no implementation details

Implementing Interfaces

- Add the name of the interface to the class declaration

```
public class Coffee : IBeverage
```

- Implement all interface members
- Use the interface type and the derived class type interchangeably

```
Coffee coffee1 = new Coffee();
IBeverage coffee2 = new Coffee();
```

The **coffee2** variable will only expose members defined by the **IBeverage** interface

Implementing Multiple Interfaces

- Add the names of each interface to the class declaration

```
public class Coffee : IBeverage, IInventoryItem
```

- Implement every member of every interface
- Use explicit implementation if two interfaces have a member with the same name

```
// This is an implicit implementation.
public bool IsFairTrade { get; set; }

//These are explicit implementations.
public bool IInventoryItem.IsFairTrade { get; }
public bool IBeverage.IsFairTrade { get; set; }
```

Practicing Interfaces

- Create an interface around something actionable, remember to start its name with an I

- Add a class that implements the interface

- Declare a variable of the interface type and assign an instance of the class.

- Declare and instantiate a List<InterfaceType> and add a couple of class instances to it.

Lab 6

IDisposable

- Used to dispose of external dependencies.
- Dispose method.

Introducing Generics

- Create classes and interfaces that include a type parameter

```
public class CustomList<T>
{
   public T this[int index] { get; set; }
   public void Add(T item) { ... }
   public void Remove(T item) { ... }
}
```

- Specify the type argument when you instantiate the class

```
CustomList<Coffee> coffees =
   new CustomList<Coffee>();
```

Advantages of Generics

Generic types offer three advantages over non-generic types:

- Type safety
- No casting
- No boxing and unboxing

Constraining Generics

You can constrain type parameters in six ways:

- where T : <name of interface>
- where T : <name of base class>
- where T : U
- where T : new()
- where T : struct
- where T : class

Using Generic List Collections

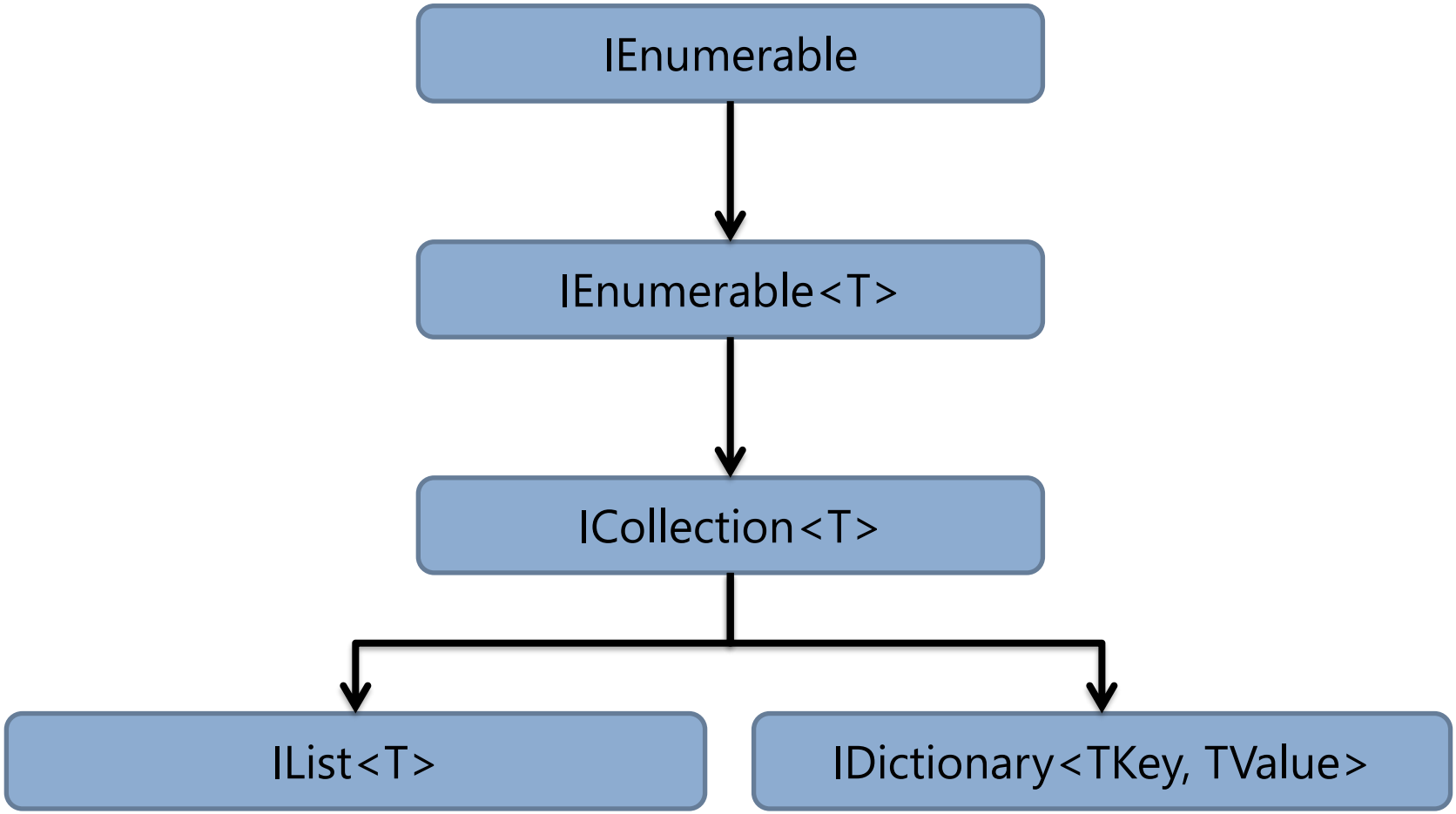
Generic list classes store collections of objects of type **T**:

- **List<T>** is a general purpose generic list
- **LinkedList<T>** is a generic list in which each item is linked to the previous item and the next item in the collection
- **Stack<T>** is a last in, first out collection
- **Queue<T>** is a first in, first out collection

Using Generic Dictionary Collections

- Generic dictionary classes store key-value pairs
- Both the key and the value are strongly typed
- **Dictionary<TKey, TValue>** is a general purpose, generic dictionary class
- **SortedList<TKey, TValue>** and **SortedDictionary<TKey, TValue>** collections are sorted by key

Using Collection Interfaces



Creating Enumerable Collections

- Implement **IEnumerable<T>** to support enumeration (**foreach**)
- Implement the **GetEnumerator** method by either:
  - Creating an **IEnumerator<T>** implementation
  - Using an iterator
- Use the **yield return** statement to implement an iterator

Practicing Generics

- Create a generic class called DataStore<T>

- Add a property Data of type T

- Create an instance of the class using int for T.
  Set the Data property

- Do the same using a string

- Create a new class DataItem with some properties

- Create an instance of the DataStore class using
  DataItem as a generic parameter. Set the Data property.