Module 4

# Creating Classes and Implementing Type-Safe Collections

# Module Overview

- Creating Classes
- Defining and Implementing Interfaces
- Implementing Type-Safe Collections

# Lesson 1: Creating Classes

- Creating Classes and Members
- Instantiating Classes
- Using Constructors
- Reference Types and Value Types
- Creating Static Classes and Members

# Creating Classes and Members

- Use the **class** keyword

```
public class DrinksMachine
{
    // Methods, fields, properties, and events.
}
```

- Specify an access modifier:
  - public
  - internal
  - private
- Add methods, fields, properties, and events

# Instantiating Classes

- To instantiate a class, use the **new** keyword

  ```
  DrinksMachine dm = new DrinksMachine();
  ```

- To infer the type of the new object, use the **var** keyword

  ```
  var dm = new DrinksMachine();
  ```

- To call members on the instance, use the dot notation

  ```
  dm.Model = "BeanCrusher 3000";
  dm.Age = 2;
  dm.MakeEspresso();
  ```

# Using Constructors

- Constructors are a type of method:
  - Share the name of the class
  - Called when you instantiate a class
- A default constructor accepts no arguments

```
public class DrinksMachine
{
    public void DrinksMachine()
    {
        // This is a default constructor.
    }
}
```

- Classes can include multiple constructors
- Use constructors to initialize member variables

# Reference Types and Value Types

- Value types
  - Contain data directly

```
int First = 100;
int Second = First;
```

  - In this case, **First** and **Second** are two distinct items in memory

- Reference types
  - Point to an object in memory

```
object First = new Object();
object Second = First;
```

  - In this case, **First** and **Second** point to the same item in memory

# Text Continuation

# Creating Static Classes and Members

- Use the static keyword to create a static class

```
public static class Conversions
{
    // Static members go here.
}
```

- Call members directly on the class name

```
double weightInKilos = 80;
double weightInPounds =
    Conversions.KilosToPounds(weightInKilos);
```

- Add static members to non-static classes

# Lesson 2: Defining and Implementing Interfaces

- Introducing Interfaces
- Defining Interfaces
- Implementing Interfaces
- Implementing Multiple Interfaces
- Implementing the IComparable Interface
- Implementing the IComparer Interface

# 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; }
```

# Implementing the IComparable Interface

- If you want instances of your class to be sortable in collections, implement the **IComparable** interface

```
public interface IComparable
{
    int CompareTo(Object obj);
}
```

- The **ArrayList.Sort** method calls the **IComparable.CompareTo** method on collection members to sort items in a collection

# Implementing the IComparer Interface

- To sort collections by custom criteria, implement the **IComparer** interface

```
public interface IComparer
{
    int Compare(Object x, Object y);
}
```

- To use an **IComparer** implementation to sort an **ArrayList**, pass an **IComparer** instance to the **ArrayList.Sort** method

```
ArrayList coffeeList = new ArrayList();
// Add some items to the collection.
coffeeList.Sort(new CoffeeRatingComparer());
```

# Lesson 3: Implementing Type-Safe Collections

- Introducing Generics
- Advantages of Generics
- Constraining Generics
- Using Generic List Collections
- Using Generic Dictionary Collections
- Using Collection Interfaces
- Creating Enumerable Collections
- Demonstration: Adding Data Validation and Type-Safety to the Application Lab

# 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
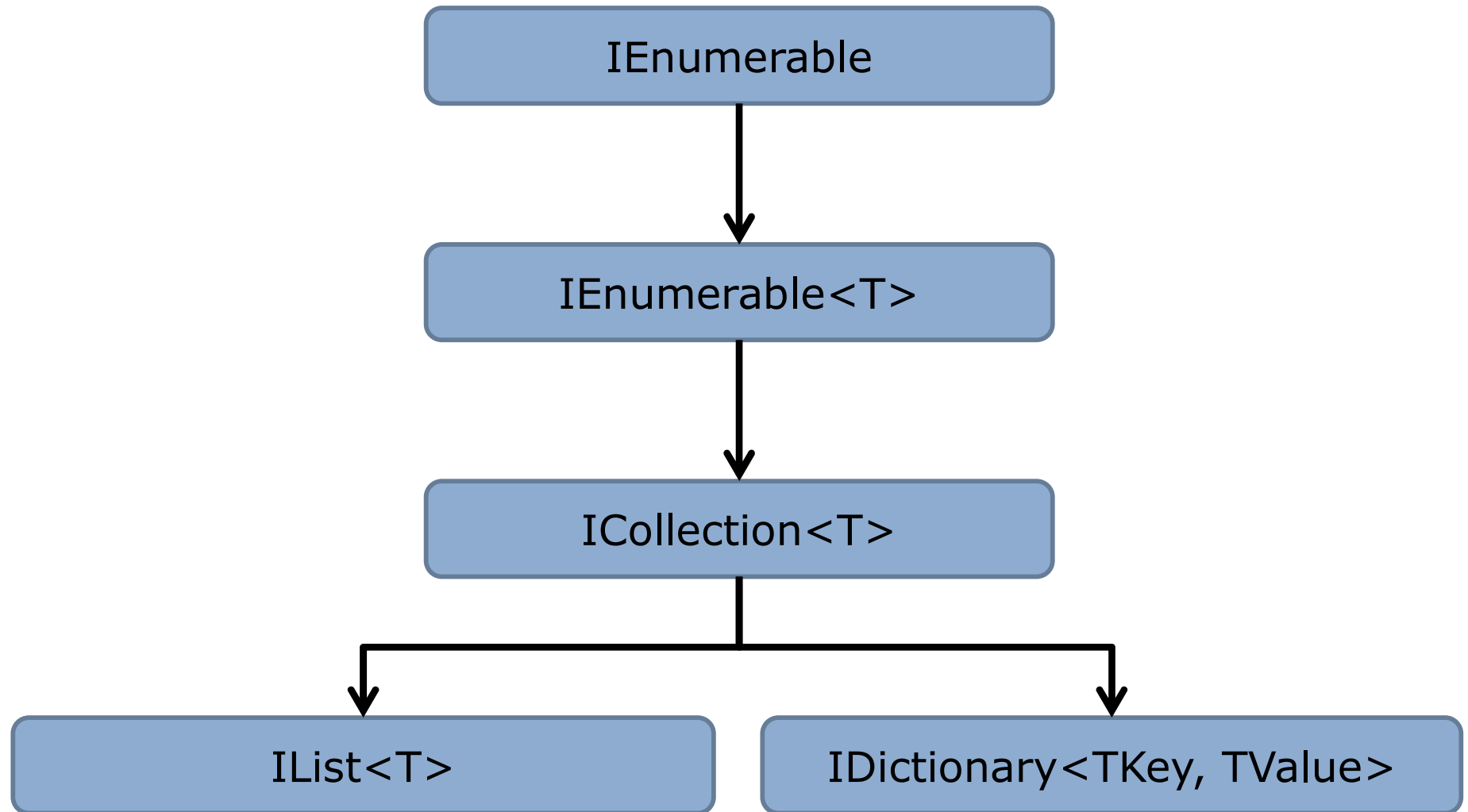
# Using Generic List Collections

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

# Text Continuation

# Text Continuation

# Lab: Adding Data Validation and Type-Safety to the Application

- Exercise 1: Implementing the Teacher, Student, and Grade Structs as Classes
- Exercise 2: Adding Data Validation to the Grade Class
- Exercise 3: Displaying Students in Name Order
- Exercise 4: Enabling Teachers to Modify Class and Grade Data

Estimated Time: 75 minutes