Generics and Collections

20.4 Type Constraints

Constraint	Description
`where T: struct`	The type argument must be a value type.
`where T : class`	The type argument must be a reference type (i.e. any class, interface, delegate, or array type.)
`where T : new()`	The type argument must have a public parameterless constructor.
`where T : <baseClass name>`	The type argument must be or derive from the specified base class.
`where T : <interface name>`	The type argument must be or implement the specified interface.

20.4 Type Constraints

Apply multiple constraints to a type parameter.

```
private static T TestMethod< T >(T x )
  where T : Employee, Iemployee,
   IComparable<T>, new()
{ // ... }
```

- Provide a comma-separated list of constraints.
- Class constraints, reference-type constraints or value-type constraints must be listed first.
- Interface constraints (if any) are listed next.
- The constructor constraint is listed last (if there is one).

20.4 Type Constraints

- Constraining multiple parameters

```
class Test{ }
private static T TestMethod< T, U >(T x, U y)
   where U : struct
   where T : Test, new()
  {
      …
  }
```

20.5 Overloading Generic Methods

- A generic method may be overloaded.
- Each overloaded method must have a unique signature.
- A generic method can be overloaded by *nongeneric* methods with the same method name.

20.6 Generic Classes

- A generic class provides a means for describing a class in a type-independent manner.

- It increases *software reusability*.

- You can use a simple, concise notation to indicate the actual type(s) that should be used in place of the class’s type parameter(s).

20.6 Generic Classes

- At compilation time, the compiler ensures your code’s type safety.

- The runtime system replaces type parameters with type arguments to enable your client code to interact with the generic class.

20.6 Generic Classes

- One generic `Stack` class

  Could be the basis for creating many `Stack` classes

  e.g., `Stack` of `double`, `int`, `char`, and `Employee`

- To define a generic class

```
class Stack < T >
{ //… }
```

20.6 Generic Classes

- IEnumerable< T > interface

  Describes the functionality of any objects that can be iterated over.

  - foreach statements iterate over any so-called IEnumerable< T > object
  - i.e. arrays, collections and LINQ queries

- Two way to pass an array to a generic method:
  - T[] array
  - IEnumerable< T > array

20.6 Generic Classes

- *Practice:*
  - Open 22-1-NongenericStack (Stack of `int`)
  - Make `Stack` class generic
  - Declare two generic methods of `TestPush` and `TestPop`

21.1 Introduction to Collections

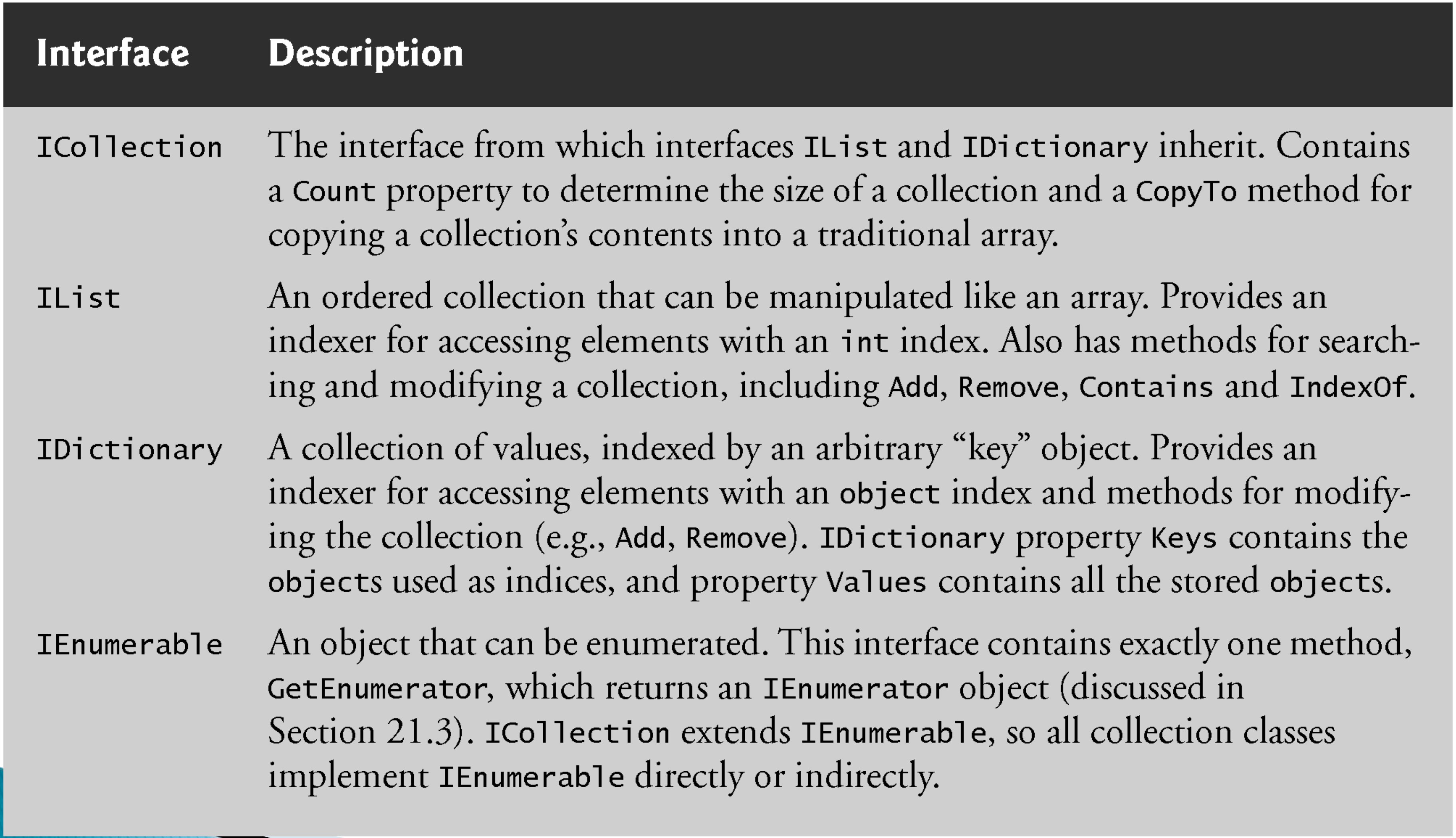
- `Collection` classes

  Are the *prepackaged* data-structure classes provided by the .NET Framework

- For the vast majority of apps, there’s no need to build such custom data structures.

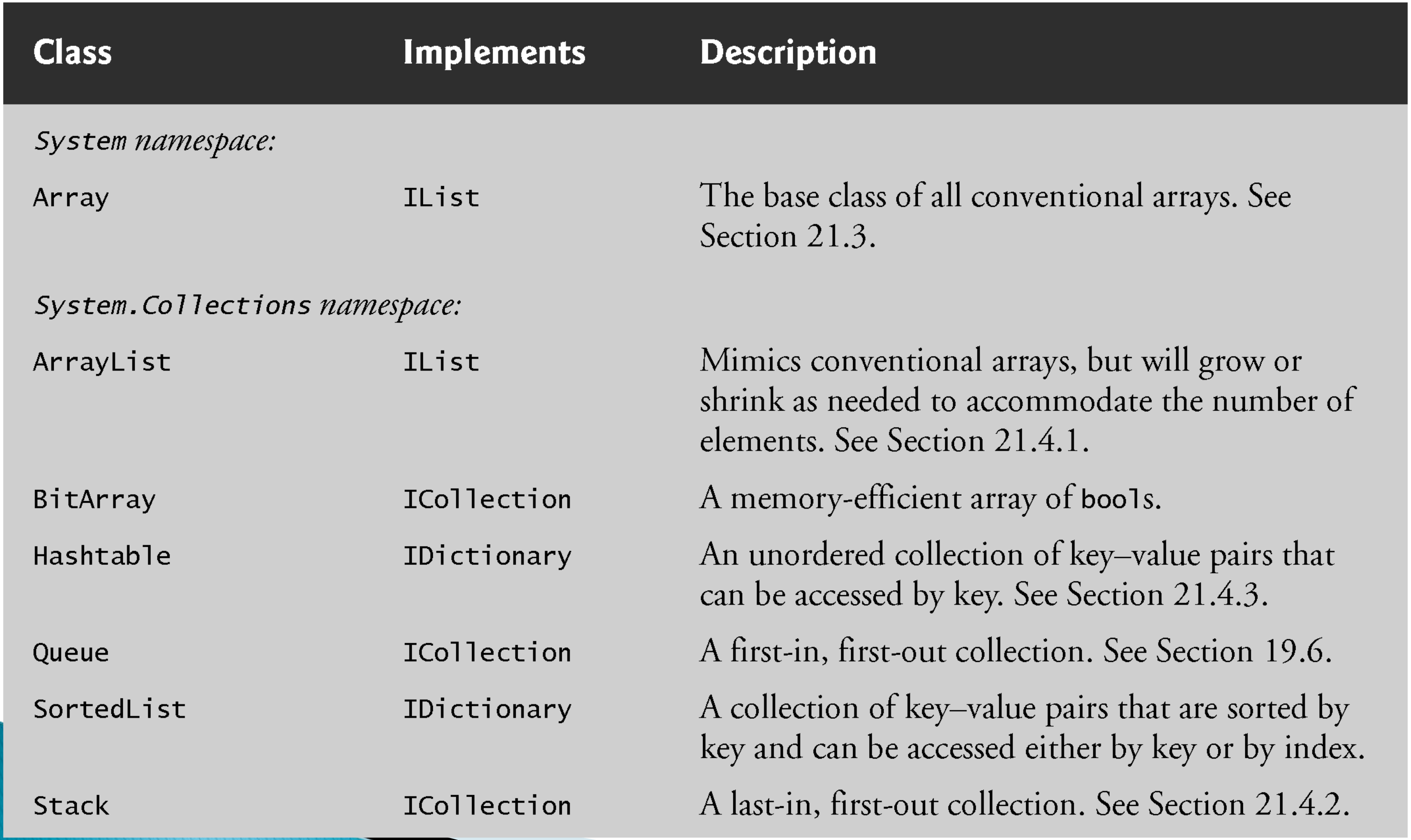
- Each instance of one of these classes is a collection of items.

21.2 Some Common Collection Interfaces



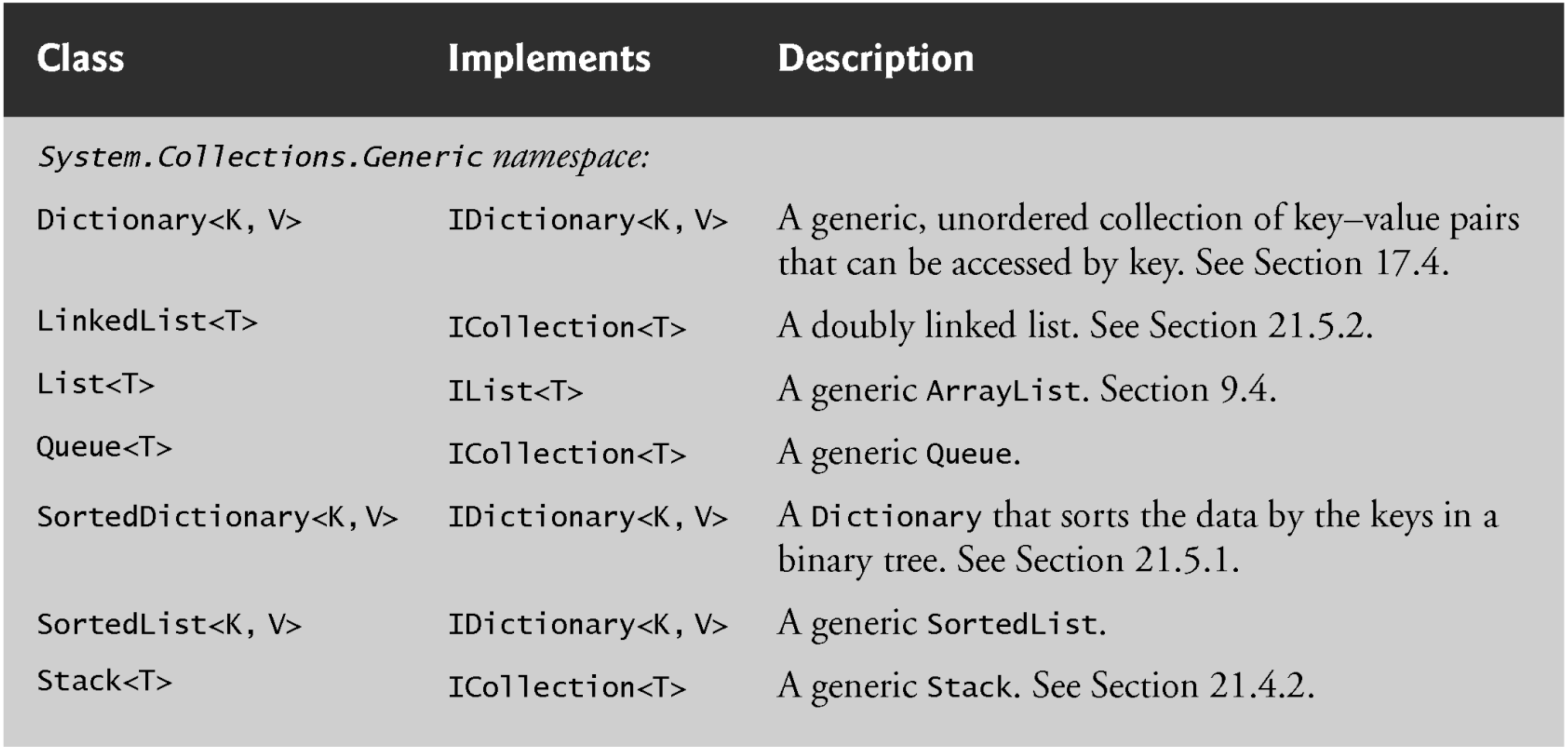
Interface	Description
ICollection	The interface from which interfaces IList and IDictionary inherit. Contains a Count property to determine the size of a collection and a CopyTo method for copying a collection's contents into a traditional array.
IList	An ordered collection that can be manipulated like an array. Provides an indexer for accessing elements with an int index. Also has methods for searching and modifying a collection, including Add, Remove, Contains and IndexOf.
IDictionary	A collection of values, indexed by an arbitrary "key" object. Provides an indexer for accessing elements with an object index and methods for modifying the collection (e.g., Add, Remove). IDictionary property Keys contains the objects used as indices, and property Values contains all the stored objects.
IEnumerable	An object that can be enumerated. This interface contains exactly one method, GetEnumerator, which returns an IEnumerator object (discussed in Section 21.3). ICollection extends IEnumerable, so all collection classes implement IEnumerable directly or indirectly.

21.1 Some Collection Classes



Class	Implements	Description
System namespace:		
Array	IList	The base class of all conventional arrays. See Section 21.3.
System.Collections namespace:		
ArrayList	IList	Mimics conventional arrays, but will grow or shrink as needed to accommodate the number of elements. See Section 21.4.1.
BitArray	ICollection	A memory-efficient array of bools.
Hashtable	IDictionary	An unordered collection of key–value pairs that can be accessed by key. See Section 21.4.3.
Queue	ICollection	A first-in, first-out collection. See Section 19.6.
SortedList	IDictionary	A collection of key–value pairs that are sorted by key and can be accessed either by key or by index.
Stack	ICollection	A last-in, first-out collection. See Section 21.4.2.

21.1 Some Collection Classes



Class	Implements	Description
System.Collections.Generic namespace:		
Dictionary<K, V>	IDictionary<K, V>	A generic, unordered collection of key–value pairs that can be accessed by key. See Section 17.4.
LinkedList<T>	ICollection<T>	A doubly linked list. See Section 21.5.2.
List<T>	IList<T>	A generic ArrayList. Section 9.4.
Queue<T>	ICollection<T>	A generic Queue.
SortedDictionary<K,V>	IDictionary<K, V>	A Dictionary that sorts the data by the keys in a binary tree. See Section 21.5.1.
SortedList<K, V>	IDictionary<K, V>	A generic SortedList.
Stack<T>	ICollection<T>	A generic Stack. See Section 21.4.2.

21.3 Class `Array` and Enumerators

- All arrays inherit implicitly from `Array`.
- All array types implement `IEnumerable` interface.

Methods and properties:

- `GetEnumerator()`

  Returns an enumerator that can *iterate* over the collection.
- `MoveNext()`

  Moves the enumerator to the next element in the collection.
- `Reset()`

  Sets the enumerator to its initial position, which is before the first element in the collection.
- `Current`

  Gets the current element in the collection.

21.3 Class `Array` and Enumerators

Note: Enumerators are “fail fast”.

- If a collection is modified after an enumerator is created for that collection, the enumerator immediately becomes invalid.

- Any methods called on the enumerator (i.e. `MoveNext` and `Reset`) after this point throw `InvalidOperationException`s.

- *Practice:* 22-2-UsingArray-Practice
  - Follow the instructions

21.4.1 Class ArrayList

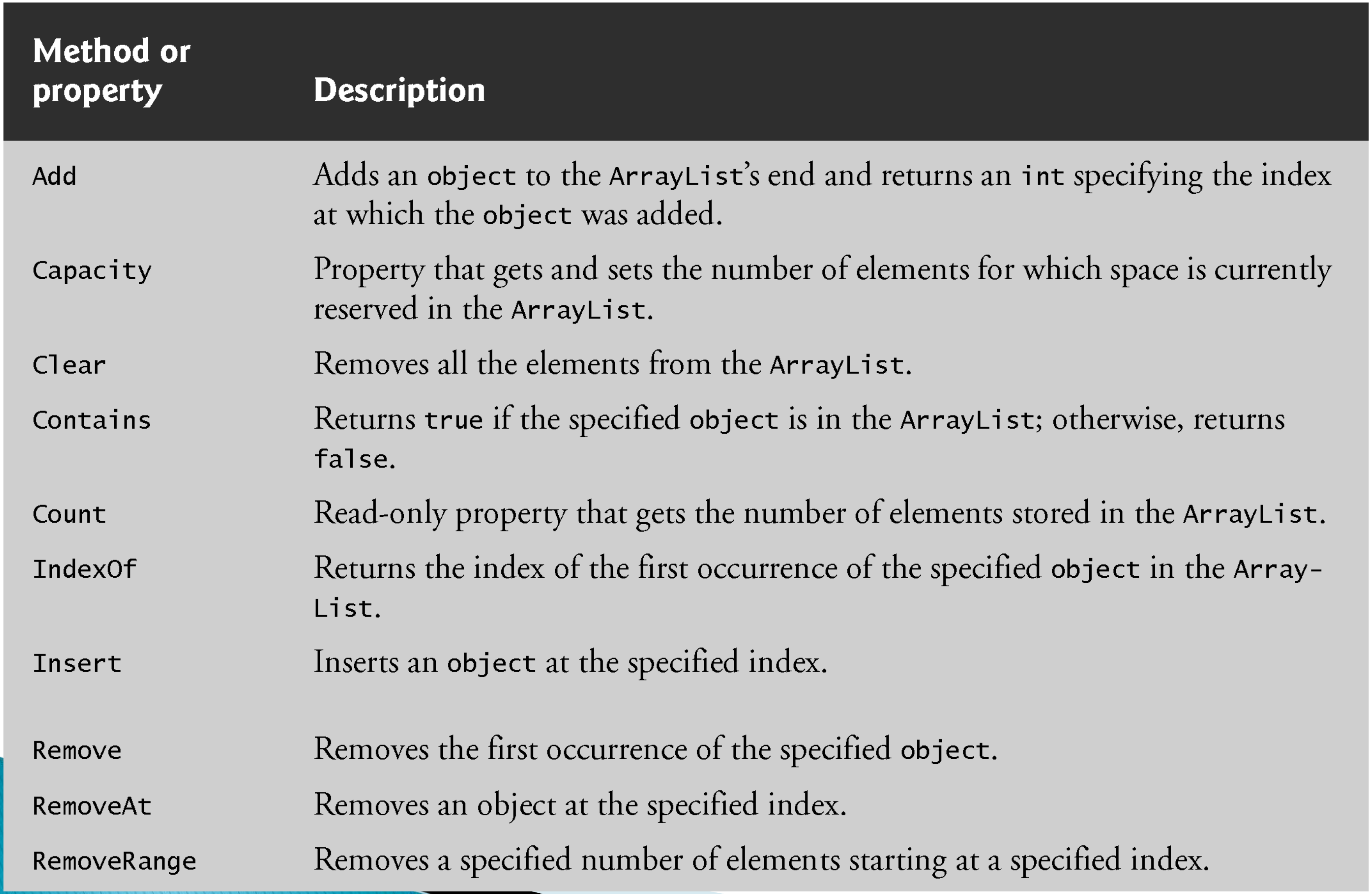
- ArrayList collection class
  - Mimics the functionality of conventional arrays.
  - Provides dynamic resizing of the collection through the class's methods.
- Property Capacity
  - The number of elements currently reserved for the ArrayList.
- List<T> class
  - The generic version of ArrayList

21.4.1 Class ArrayList

- Inserting additional elements into an ArrayList whose current size is less than is capacity is a *fast* operation.

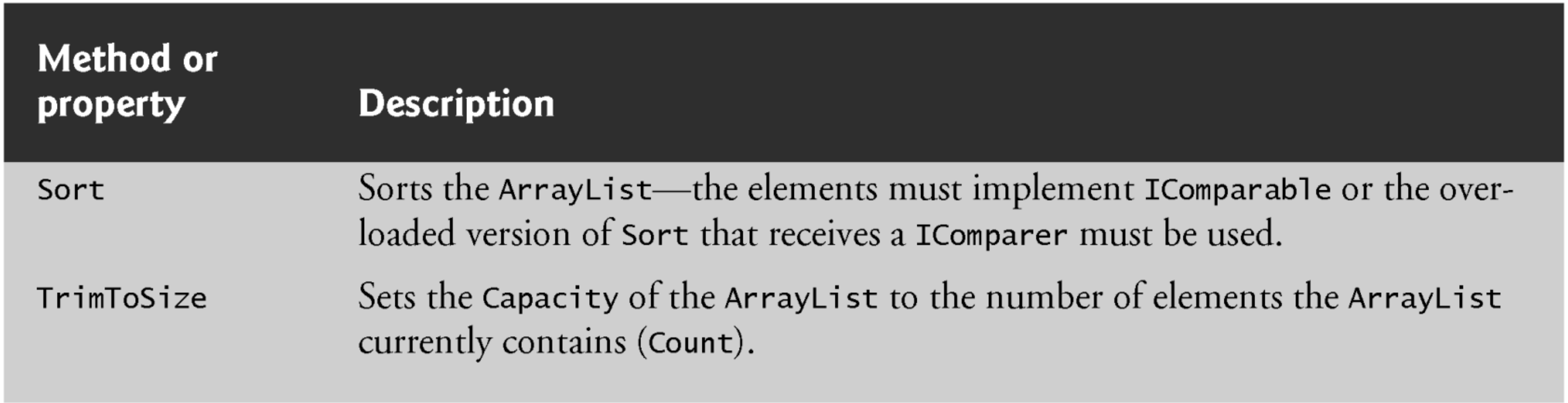
- It’s a *slow* operation to insert an element into an ArrayList that needs to grow larger to accommodate a new element.
  - An ArrayList that’s at its capacity must have its memory reallocated and the existing values copied into it.

21.4.1 Class ArrayList



Method or property	Description
Add	Adds an object to the ArrayList's end and returns an int specifying the index at which the object was added.
Capacity	Property that gets and sets the number of elements for which space is currently reserved in the ArrayList.
Clear	Removes all the elements from the ArrayList.
Contains	Returns true if the specified object is in the ArrayList; otherwise, returns false.
Count	Read-only property that gets the number of elements stored in the ArrayList.
IndexOf	Returns the index of the first occurrence of the specified object in the ArrayList.
Insert	Inserts an object at the specified index.
Remove	Removes the first occurrence of the specified object.
RemoveAt	Removes an object at the specified index.
RemoveRange	Removes a specified number of elements starting at a specified index.

21.4.1 Class `ArrayList`



Method or property	Description
`Sort`	Sorts the `ArrayList`—the elements must implement `IComparable` or the overloaded version of `Sort` that receives a `IComparer` must be used.
`TrimToSize`	Sets the `Capacity` of the `ArrayList` to the number of elements the `ArrayList` currently contains (`Count`).

- *Practice:* 22-3-ArrayListTest-Practice
  - Follow the instructions

21.4.1 Class `ArrayList`

- Methods `IndexOf` and `Contains`
  - Each perform a linear search
  - It is a costly operation for large `ArrayList`s

- Method `BinarySearch`
  - returns the index of the element, or a negative number if the element is not found.
  - If the `ArrayList` is sorted, use `BinarySearch` to perform a more efficient search.

21.4.1 Class `ArrayList`

- Use the indexer [] to obtain each of an `ArrayList`’s elements.

- Method `Remove`
  - This method deletes a specified item from an `ArrayList` by performing a *linear search* and removing (only) the first occurrence of the specified object.
  - All subsequent elements *shift* toward the beginning of the `ArrayList` to fill the emptied position.