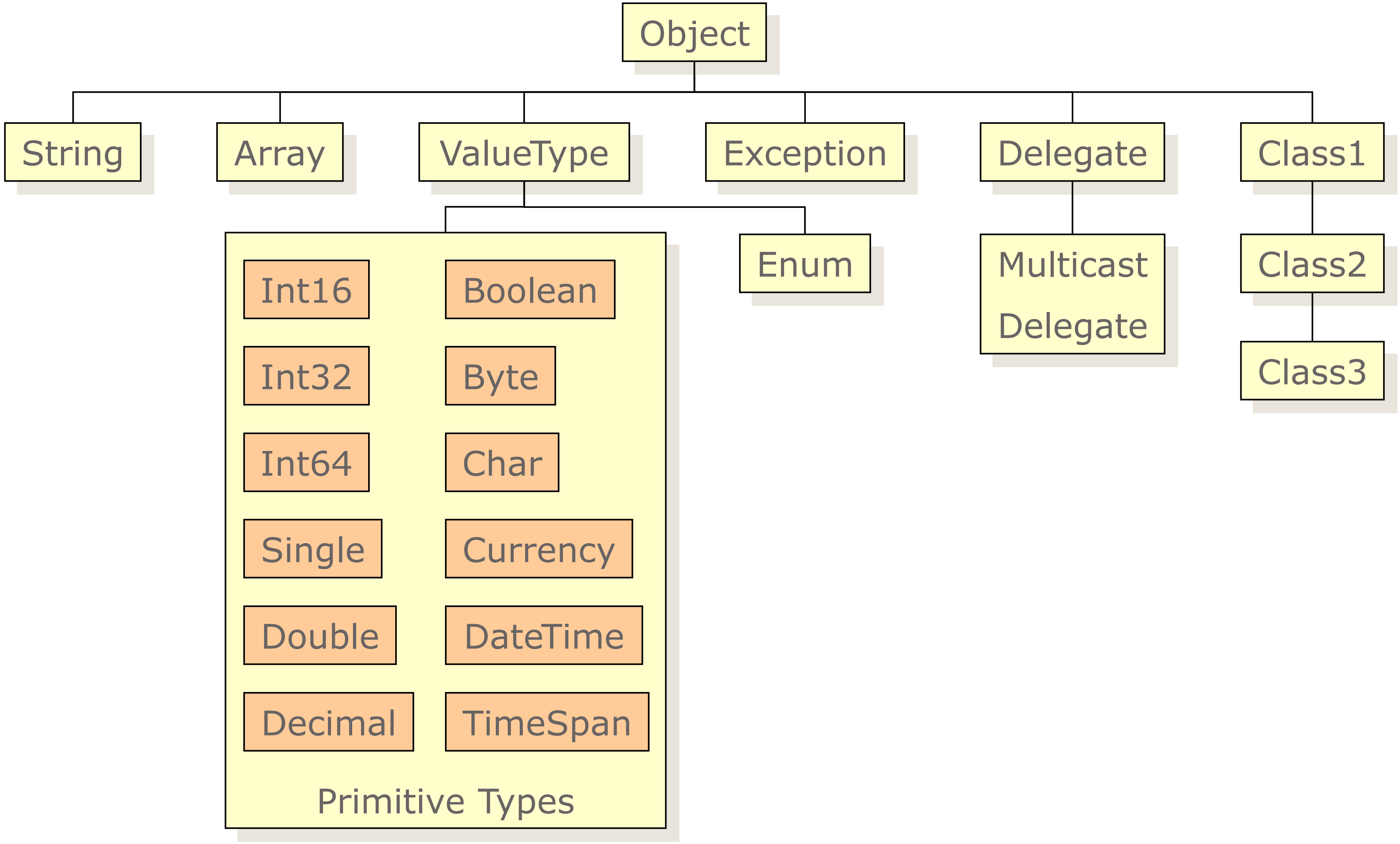
INFORMATION PROCESSING TECHNIQUES

Week 03

Murtaza Munawar Fazal

THE COMMON TYPE SYSTEM



STRUCTS & CLASSES IN C#

- In C# structs are very different from classes. Structs in C# are designed to encapsulate lightweight objects. They are value types (not reference types), so they're passed by value.
- They are sealed, which means they cannot be derived from or have any base class.
- Classes in C# are different from classes in C++ in the following ways:
  - There is no access modifier on the name of the base class and inheritance is always public.
  - A class can only be derived from one base class. If no base class is explicitly specified, then the class will automatically be derived from System.Object.
  - In C++, the only types of class members are variables, functions, constructors, destructors and operator overloads, C# also permits delegates, events and properties.
  - The access modifiers public, private and protected have the same meaning as in C++ but there are three additional access modifiers available: (a) Internal (b) Protected internal (c) Private Protected

INTERFACES

C# does not support Multiple Inheritance

However a class can implement number of interfaces

It contains methods, properties, indexers, and events

```
interface DataBind
{
    void Bind(IDataBinder bind);
}
Class EditBox: Control, DataBind
{
    void DataBind.Bind(IDataBinder bind) {…}
}
```

VIRTUAL METHODS

- In C# one can choose to override a virtual function from base class. Derived method can participate in polymorphism only if it uses the keyword override before it.
- In C++, if provided the same syntax method in derived class as base class virtual method, it will be automatically be overridden.
- In C# we have abstract methods and in C++ pure virtual methods. Both may not be exactly same, but are equivalent (as pure virtual can have function body)
- EXAMPLE:

```
class Base{
        public virtual string VirtualMethod()
                        { return "base virtual"; }
}
class Derived : Base{
        public override string VirtualMethod()
        { return "Derived overriden"; }
}
```

VALUE TYPE V/S REFERENCE TYPE

- VALUE TYPE:
  - Instances of value types do not have referential identity nor referential comparison semantics i.e. equality and inequality comparisons for value types compare the actual data values within the instances, unless the corresponding operators are overloaded.
  - Value types are derived from System.ValueType, always have a default value, and can always be created and copied.
  - They cannot derive from each other (but can implement interfaces) and cannot have an explicit default (parameterless) constructor.

VALUE TYPE V/S REFERENCE TYPE

- REFERENCE TYPE:
  - Reference types are type-safe object pointers. Allocated in the “managed heap”
  - Reference types have the notion of referential identity - each instance of a reference type is inherently distinct from every other instance, even if the data within both instances is the same.
  - It is not always possible to create an instance of a reference type, nor to copy an existing instance, or perform a value comparison on two existing instances.
  - Specific reference types can provide services by exposing a public constructor or implementing a corresponding interface (such as ICloneable or IComparable). Examples: System.String, System.Array
  - Four kinds of reference types: Classes, arrays, delegates, and interfaces.
    - When instances of value types go out of scope, they are instantly destroyed and memory is reclaimed.
    - When instances of reference types go out of scope, they are garbage collected.

BOXING AND UNBOXING

- Boxing is the operation of converting a value-type object into a value of a corresponding reference type.
- Boxing in C# is implicit.
- Unboxing is the operation of converting a value of a reference type (previously boxed) into a value of a value type.
- Unboxing in C# requires an explicit type cast. A boxed object of type T can only be unboxed to a T (or a nullable T).
- EXAMPLE :
  - int box_var = 42; // Value type.
  - object bar = box_var; // foo is boxed to bar.
  - int box_var2 = (int)bar; // Unboxed back to value type.

27.3 Class Array and Enumerators

- Class `Array`
  - All arrays implicitly inherit from this abstract base class
    - Defines property `Length`
      - Specifies the number of elements in the array
    - Provides `static` methods that provide algorithms for processing arrays
      - For a complete list of class `Array` 's methods visit:

        msdn2.microsoft.com/en-us/library/system.array.aspx

COMMON DATA STRUCTURES

- We’ve seen Array only so far → fixed-size (can grow with Resize)
- Dynamic data structures can automatically grow and shrink at execution time.
- **Linked lists** are collections of data items that are “chained together”.
- **Stacks** have insertions and deletions made at only one end: the top.
- **Queues** represent waiting lines; insertions are made at the back and deletions are made from the front.
- **Binary trees** facilitate high-speed searching and sorting of data.

COLLECTIONS

- For the vast majority of applications, there is no need to build custom data structures.
- Instead, you can use the prepackaged data-structure classes provided by the .NET Framework.
- These classes are known as **collection classes**—they store collections of data. Each instance of one of these classes is a **collection** of items.
- Collection classes enable programmers to store sets of items by using existing data structures, without concern for how they are implemented.
- `System.Collections` contains collections that store references to objects.

ARRAYLIST

- Class `ArrayList`
  - Mimics the functionality of conventional arrays
  - Provides dynamic resizing of the collection through the class's methods
    - Property `Capacity`
      - Manipulate the capacity of the `ArrayList`
      - When `ArrayList` needs to grow, it by default doubles its `Capacity`
  - Store references to objects
    - All classes derive from class `object`
      - Can contain objects of any type

ARRAYLIST

Method / Property	Description
Add	Adds an object to the end of the ArrayList.
Capacity	Property that gets and sets the number of elements for which space is currently reserved in the ArrayList.
Clear	Removes all elements from the ArrayList.
Contains	Determines whether an element is in the ArrayList.
Count	Read-only property that gets the number of elements stored in the ArrayList.
IndexOf	Returns the zero-based index of the first occurrence of a value in the ArrayList
Insert	Inserts an element into the ArrayList at the specified index.
Remove	Removes the first occurrence of a specific object from the ArrayList.
RemoveAt	Removes the element at the specified index of the ArrayList.
TrimToSize	Sets the capacity to the actual number of elements in the ArrayList.

- The **ArrayList** collection class is a conventional arrays and provides dynamic resizing of the collection.

STACK & QUEUE

- Stack:
  - Push
  - Pop
  - Peek

- Queue:
  - Enqueue
  - Dequeue
  - Peek

STACK

- Class Stack
  - Contains methods Push and Pop to perform the basic stack operations
  - Method Push
    - Takes and inserts an object to the top of the Stack
    - Grows to accommodate more objects
      - When the number of items on the Stack (the Count property) is equal to the capacity at the time of the Push operation
    - Can store only references to objects
      - Value types are implicitly boxed before they are added
  - Method Pop
    - Removes and returns the object current on the top of the Stack
  - Method Peek
    - Returns the value of the top stack element
    - Does not remove the element from the Stack
  - Note on methods Pop and Peek
    - Throws InvalidOperationException if the Stack is empty
  - Property Count
    - Obtain the number of elements in Stack

```
// Fig. 27.6: StackTest.cs
// Demonstrating class Stack.
using System;
using System.Collections;

public class StackTest
{
   public static void Main( string[] args )
   {
      Stack stack = new Stack(); // default Capacity of 10

      // create objects to store in the stack
      bool aBoolean = true;
      char aCharacter = '$';
      int anInteger = 34567;
      string aString = "hello";

      // use method Push to add items to (the top of) the stack
      stack.Push( aBoolean );
      PrintStack( stack );
      stack.Push( aCharacter );
      PrintStack( stack );
      stack.Push( anInteger );
      PrintStack( stack );
      stack.Push( aString );
      PrintStack( stack );

      // check the top element of the stack
      Console.WriteLine( "The top element of the stack is {0}\n",
         stack.Peek() );
```

Create a stack with the default initial capacity of 10 elements

Add four elements to the stack

Obtain the value of the top stack element

Outline

StackTest.cs

(1 of 3)

```

      // remove items from stack
      try
      {
         while ( true )
         {
            object removedObject = stack.Pop();
            Console.WriteLine( removedObject + " popped" );
            PrintStack( stack );
         } // end while
      } // end try
      catch ( InvalidOperationException exception )
      {
         // if exception occurs, print stack trace
         Console.Error.WriteLine( exception );
      } // end catch
   } // end Main

   // print the contents of a stack
   private static void PrintStack( Stack stack )
   {
      if ( stack.Count == 0 )
         Console.WriteLine( "stack is empty\n" ); // the stack is empty
      else
      {
         Console.Write( "The stack is: " );

         // iterate through the stack with a foreach statement
         foreach ( object element in stack )
            Console.Write( "{0} ", element ); // invokes ToString
```

Obtain and remove the value of the top stack element

Use `foreach` statement to iterate over the stack and output its contents

StackTest.cs

(2 of 3)

```

         Console.WriteLine( "\n" );
      } // end else
   } // end method PrintStack
} // end class StackTest
```

```
The stack is: True

The stack is: $ True

The stack is: 34567 $ True

The stack is: hello 34567 $ True

The top element of the stack is hello

hello popped
The stack is: 34567 $ True

34567 popped
The stack is: $ True

$ popped
The stack is: True

True popped
stack is empty

System.InvalidOperationException: Stack empty.
   at System.Collections.Stack.Pop()
   at StackTest.Main(String[] args) in C:\examples\ch27\
      fig27_06\StackTest\StackTest.cs:line 37
```

Outline

StackTest.cs

(3 of 3)

COMMON PROGRAMMING ERROR

- Attempting to `Peek` or `Pop` an empty `Stack` (a `Stack` whose `Count` property is 0) causes an `InvalidOperationException`.

HASHTABLE

- Arrays uses nonnegative integer indexes as keys. Sometimes associating these integer keys with objects to store them is impractical, so we develop a scheme for using arbitrary keys.
- When an application needs to store something, the scheme could convert the application **key** rapidly to an **index**.
- Once the application has a **key** for which it wants to retrieve the data, simply apply the conversion to the key to find the array **index** where the data resides.
- The scheme we describe here is the basis of a technique called **hashing**, in which we store data in a data structure called a **hash table**.

HASHTABLE

- A hash function performs a calculation that determines where to place data in the hash table.
- The hash function is applied to the key in a key/value pair of objects.
- Class `Hashtable` can accept any object as a key. For this reason, class object defines method `GetHashCode`, which all objects inherit.

The load factor

- The ratio of the number of objects stored in the hash table to the total number of cells of the hash table
- Affects the performance of hashing schemes
  - The chance of collisions tends to increase as this ratio gets higher

HASHING

- Collisions
  - Problem: Two different keys "hash into" the same cell in the array
    - Solution 1: "Hash Again"
      - Hashing process is designed to be quite random
        - Assumption that within a few hashes, an available cell will be found
    - Solution 2: Uses one hash to locate the first candidate cell. If the cell is occupied, successive cells are searched linearly until an available cell is found
      - Retrieval works the same way
        - The key is hashed once, the resulting cell is checked to determine whether it contains the desired data

          - If it does, the search is complete

          - If it does not, successive cells are searched linearly until the desired data is found
    - Solution 3: Have each cell of the table be a hash "bucket" of all the key–value pairs that hash to that cell
      - Typically as a linked list
      - .NET Framework's `Hashtable` class implements this solution

HASHTABLE

- Hashtable method `ContainsKey` determines whether a key is in the hash table.
- Read-only property `Keys` returns an `ICollection` that contains all the keys.
- Hashtable property `Count` returns the number of key/value pairs in the Hashtable.
- If you use a foreach statement with a Hashtable object, the iteration variable will be of type `DictionaryEntry`.
- The enumerator of a Hashtable (or any other class that implements IDictionary) uses the DictionaryEntry structure to store key/value pairs.
- This structure provides properties Key and Value for retrieving the key and value of the current element.
- If you do not need the key, class Hashtable also provides a read-only Values property that gets an `ICollection` of all the values stored in the Hashtable.

HASHTABLE

- Class `Hashtable`
  - Method `ContainsKey`
    - Determine whether the word is in the hash table
  - Property `Keys`
    - Get an `ICollection` that contains all the keys in the hash table
  - Property `Value`
    - Gets an `ICollection` of all the values stored in the hash table
  - Property `Count`
    - Get the number of key-value pairs in the hash table

PERFORMANCE TIP

- The load factor in a hash table is a classic example of a space/time trade-off: By increasing the load factor, we get better memory utilization, but the application runs slower due to increased hashing collisions. By decreasing the load factor, we get better application speed because of reduced hashing collisions, but we get poorer memory utilization because a larger portion of the hash table remains empty.

COMMON PROGRAMMING ERROR

- Using the `Add` method to add a key that already exists in the hash table causes an `ArgumentException`.

PROBLEMS WITH NON-GENERIC COLLECTIONS

- `Hashtable` stores its keys and data as `object` references
  - Say we store only `string` keys and `int` values by convention
    - Inefficient!
- Cannot control what is being put into the `Hashtable`
  - `InvalidCastException` might be thrown at execution time

WHY DO WE NEED GENERICS?

- Another method of software re-use.
- When we implement an algorithm, we want to re-use it for different **types**.
- Example: We write a **generic** method for sorting an array of objects, then call the **generic** method with an array of any **type**.
- The compiler performs **type checking** to ensure that the array passed to the sorting method contains only elements of the same **type**.
- Generics provide compile-time type safety.

GENERICS

- Generics use type parameters, which make it possible to design classes and methods that do not specify the type used until the class or method is instantiated.
- The main advantage is that one can use generic type parameters to create classes and methods that can be used without incurring the cost of runtime casts or boxing operations.
- EXAMPLE:

```
public class GenericList<T>
{
        void Add(T input) { }
}
class TestGenericList
{
        private class ExampleClass { }
        static void Main() {
        // Declare a list of type int.
        GenericList<int> list1 = new GenericList<int>();
        // Declare a list of type string.
        GenericList<string> list2 = new GenericList<string>();
        }
}
```

GENERIC METHODS

- Generic methods enable you to specify, with a single method declaration, a set of related methods.

```
private static void DisplayArray( T[] inputArray )
{
       foreach ( T element in inputArray )
          Console.Write( element + " " );

       Console.WriteLine( "\n" );
}
```

- However, it will not compile, because its syntax is not correct.

GENERIC METHODS

- All generic method declarations have a **type-parameter list** delimited by angle brackets that follows the method's name.
- Each type-parameter list contains one or more **type parameters**.
- A type parameter is an identifier that is used in place of actual type names.
- The type parameters can be used to declare the return type, the parameter types and the local variable types in a generic method declaration.
- Type parameters act as placeholders for **type arguments** that represent the types of data that will be passed to the generic method.
- A generic method's body is declared like that of any other method.
- The type-parameter names throughout the method declaration must match those declared in the type-parameter list.
- A type parameter can be declared only once in the type-parameter list but can appear more than once in the method's parameter list.
- You can also use **explicit type arguments** to indicate the exact type that should be used to call a generic function, as in

```
DisplayArray< int >( intArray );
```

GENERIC COLLECTIONS

Problems with Nongeneric Collections

- Having to store data as object references causes less efficient code due to unboxing.

- The .NET Framework also includes the `System.Collections.Generic` namespace, which uses C#’s generics capabilities.
- Many of these new classes are simply generic counterparts of the classes in namespace **`System.Collections`**.
- Generic collections eliminate the need for explicit type casts that decrease type safety and efficiency.
- Generic collections are especially useful for storing structs, since they eliminate the overhead of boxing and unboxing.

FEW GENERIC COLLECTION CLASSES

- List(T)
- Stack(T)
- Queue(T)
- LinkedList(T)
- SortedList(TKey, TValue)

LIST<T>

- Arrays have problem that you must know how many elements you want in advance
  - This is not always known
- List class is collection with variable size
  - Dynamically increases in size if needed
  - When an array reaches its capacity, need to create new array, and copy all elements from old array to new array
    - Ugh!

CREATING A LIST

List<type> listname

Example:

```
List<string> stringList = new List<string>();    // Create list of string. Don’t forget ()

stringList.Add ( “Quick” );
stringList.Add ( “Brown” );
stringList.Add ( “Fox” );
```

- Add elements with Add() method
- Clear() removes all elements from list
- Remove() removes first element from list
- Sort() sorts the list
- Count property: number of elements in list

SORTEDDICTIONARY

- Generic Class `SortedDictionary`
    - Dictionary
        - A general term for a collection of key–value pairs
            - A hash table is one way to implement a dictionary
    - Does not use a hash table
        - Stores its key–value pairs in a binary search tree
            - Entries are sorted in the tree by key
                - Using the `IComparable` interface
            - Use the same `public` methods, properties and indexers with classes `Hashtable` and `SortedDictionary`
            - Takes two type arguments delimited by < >
                - The first specifies the type of key
                - The second specifies the type of value

```
// Fig. 27.8: SortedDictionaryTest.cs
// Application counts the number of occurrences of each word in a string
// and stores them in a generic sorted dictionary.
using System;
using System.Text.RegularExpressions;
using System.Collections.Generic;

public class SortedDictionaryTest
{
   public static void Main( string[] args )
   {
      // create sorted dictionary based on user input
      SortedDictionary< string, int > dictionary = CollectWords();

      // display sorted dictionary content
      DisplayDictionary( dictionary );
   } // end method Main

   // create sorted dictionary from user input
   private static SortedDictionary< string, int > CollectWords()
   {
      // create a new sorted dictionary
      SortedDictionary< string, int > dictionary =
         new SortedDictionary< string, int >();

      Console.WriteLine( "Enter a string: " ); // prompt for user input
      string input = Console.ReadLine(); // get input

      // split input text into tokens
      string[] words = Regex.Split( input, @"\s+" );
```

Namespace that contains class `SortedDictionary`

Create a dictionary of `int` values keyed with `strings`

Divide the user’s input by its whitespace characters

Outline

SortedDictionaryTest.cs

(1 of 3)

```

      // processing input words
      foreach ( string word in words )
      {
         string wordKey = word.ToLower(); // get word in lowercase

         // if the dictionary contains the word
         if ( dictionary.ContainsKey( wordKey ) )
         {
            ++dictionary[ wordKey ];
         } // end if
         else
            // add new word with a count of 1 to the dictionary
            dictionary.Add( wordKey, 1 );
      } // end foreach

      return dictionary;
   } // end method CollectWords

   // display dictionary content
   private static void DisplayDictionary< K, V >(
      SortedDictionary< K, V > dictionary )
   {
      Console.WriteLine( "\nSorted dictionary contains:\n{0,-12}{1,-12}",
         "Key:", "Value:" );
```

Convert each word to lowercase

Determine if the word is in the dictionary

Use indexer to obtain and set the key’s associated value

Create a new entry in the dictionary and set its value to 1

Modified to be completely generic; takes type parameters K and V

SortedDictionaryTest.cs

(2 of 3)

```

      // generate output for each key in the sorted dictionary
      // by iterating through the Keys property with a foreach statement
      foreach ( K key in dictionary.Keys )
         Console.WriteLine( "{0,-12}{1,-12}", key, dictionary[ key ] );

      Console.WriteLine( "\nsize: {0}", dictionary.Count );
   } // end method DisplayDictionary
} // end class SortedDictionaryTest
```

```
Enter a string:
We few, we happy few, we band of brothers

Sorted dictionary contains:
Key:        Value:
band        1
brothers    1
few,        2
happy       1
of          1
we          3

size: 6
```

Outline

SortedDictionaryTest.cs

Get an `ICollection` that contains all the keys

Output the number of different words

Iterate through the dictionary and output its elements

PERFORMANCE TIP 27.7

- Because class `SortedDictionary` keeps its elements sorted in a binary tree, obtaining or inserting a key–value pair takes O(log n) time, which is fast compared to linear searching then inserting.

COMMON PROGRAMMING ERROR 27.5

- Invoking the `get` accessor of a `SortedDictionary` indexer with a key that does not exist in the collection causes a `KeyNotFoundException`. This behavior is different from that of the `Hashtable` indexer's `get` accessor, which would return `null`.

LINKEDLIST<T>

- Generic Class LinkedList
  - Doubly-linked list
  - Each node contains:
    - Property Value
      - Matches LinkedList's single type parameter
        - Contains the data stored in the node
    - Read-only property Previous
      - Gets a reference to the preceding node (or null if the node is the first of the list)
    - Read-only property Next
      - Gets a reference to the subsequent reference (or null if the node is the last of the list)
  - Method AddLast
    - Creates a new LinkedListNode
    - Appends this node to the end of the list
  - Method AddFirst
    - Inserts a node at the beginning of the list
  - Method Find
    - Performs a linear search on the list
    - Returns the first node that contains a value equal to the passed argument
      - Returns null if the value is not found
  - Method Remove
    - Splices that node out of the LinkedList
    - Fixes the references of the surrounding nodes
  - One LinkedListNode cannot be a member of more than one LinkedList
    - Generates an InvalidOperationException

```
// Fig. 27.9: LinkedListTest.cs
// Using LinkedLists.
using System;
using System.Collections.Generic;

public class LinkedListTest
{
   private static readonly string[] colors = { "black", "yellow",
      "green", "blue", "violet", "silver" };
   private static readonly string[] colors2 = { "gold", "white",
      "brown", "blue", "gray" };

   // set up and manipulate LinkedList objects
   public static void Main( string[] args )
   {
      LinkedList< string > list1 = new LinkedList< string >();

      // add elements to first linked list
      foreach ( string color in colors )
         list1.AddLast( color );
```

Outline

LinkedListTest.cs

Declare two arrays of strings

Create a generic LinkedList of type string

Create and append nodes of array color's elements to the end of the linked list

Outline

```
      // add elements to second linked list via constructor
      LinkedList< string > list2 = new LinkedList< string >( colors2 );

      Concatenate( list1, list2 ); // concatenate list2 onto list1
      PrintList( list1 ); // print list1 elements

      Console.WriteLine( "\nConverting strings in list1 to uppercase\n" );
      ToUppercaseStrings( list1 ); // convert to uppercase string
      PrintList( list1 ); // print list1 elements

      Console.WriteLine( "\nDeleting strings between BLACK and BROWN\n" );
      RemoveItemsBetween( list1, "BLACK", "BROWN" );

      PrintList( list1 ); // print list1 elements
      PrintReversedList( list1 ); // print list in reverse order
   } // end method Main

   // output list contents
   private static void PrintList< E >( LinkedList< E > list )
   {
      Console.WriteLine( "Linked list: " );

      foreach ( E value in list )
         Console.Write( "{0} ", value );

      Console.WriteLine();
   } // end method PrintList
```

Use overloaded constructor to create a new `LinkedList` initialized with the contents of array `color2`

(2 of 5)

The generic method iterates and outputs the values of the `LinkedList`

```csharp

   // concatenate the second list on the end of the first list
   private static void Concatenate< E >( LinkedList< E > list1,
      LinkedList< E > list2 )
   {
      // concatenate lists by copying element values
      // in order from the second list to the first list
      foreach ( E value in list2 )
         list1.AddLast( value ); // add new node
   } // end method Concatenate

   // locate string objects and convert to uppercase
   private static void ToUppercaseStrings( LinkedList< string > list )
   {
      // iterate over the list by using the nodes
      LinkedListNode< string > currentNode = list.First;

      while ( currentNode != null )
      {
         string color = currentNode.Value; // get value in node
         currentNode.Value = color.ToUpper(); // convert to uppercase

         currentNode = currentNode.Next; // get next node
      } // end while
   } // end method ToUppercaseStrings
```

Outline

Append each value of `list2` to the end of `list1`

`LinkedListTest.cs`

(3 of 5)

Takes in a `LinkedList` of type `string`

Property to obtain the first `LinkedListNode`

Convert each of the `string`s to uppercase

Traverse to the next `LinkedListNode`

Outline

LinkedListTest.cs

(4 of 5)

```csharp

   // delete list items between two given items
   private static void RemoveItemsBetween< E >( LinkedList< E > list,
      E startItem, E endItem )
   {
      // get the nodes corresponding to the start and end item
      LinkedListNode< E > currentNode = list.Find( startItem );
      LinkedListNode< E > endNode = list.Find( endItem );

      // remove items after the start item
      // until we find the last item or the end of the linked list
      while ( ( currentNode.Next != null ) &&
         ( currentNode.Next != endNode ) )
      {
         list.Remove( currentNode.Next ); // remove next node
      } // end while
   } // end method RemoveItemsBetween

   // print reversed list
   private static void PrintReversedList< E >( LinkedList< E > list )
   {
      Console.WriteLine( "Reversed List:" );
```

Obtain the “boundaries” nodes of the range

Remove one element node at a time and fix the references of the surrounding nodes

```
      // iterate over the list by using the nodes
      LinkedListNode< E > currentNode = list.Last;

      while ( currentNode != null )
      {
         Console.Write( "{0} ", currentNode.Value );
         currentNode = currentNode.Previous; // get previous node
      } // end while

      Console.WriteLine();
   } // end method PrintReversedList
} // end class LinkedListTest
```

Property to obtain the last `LinkedListNode`

Traverse to the previous `LinkedListNode`

```
Linked list:
black yellow green blue violet silver gold white brown blue gray

Converting strings in list1 to uppercase

Linked list:
BLACK YELLOW GREEN BLUE VIOLET SILVER GOLD WHITE BROWN BLUE GRAY

Deleting strings between BLACK and BROWN

Linked list:
BLACK BROWN BLUE GRAY
Reversed List:
GRAY BLUE BROWN BLACK
```

`LinkedListTest.cs`

(5 of 5)

SYNCHRONIZED COLLECTIONS

- Synchronization with Collections
  - Most non-generic collections are unsynchronized
    - Concurrent access to a collection by multiple threads may cause errors
  - Synchronization wrappers
    - Prevent potential threading problems
    - Used for many of the collections that might be accessed by multiple threads
    - Wrapper object receives method calls, adds thread synchronization, and passes the calls to the wrapped collection object
  - Most of the non-generic collection classes provide `static` method `Synchronized`
    - Returns a synchronized wrapping object for the specified object

      ```
      ArrayList notSafeList = new ArrayList();
      ArrayList threadSafeList = ArrayList.Synchronized( notSafeList );
      ```

  - The collections in the .NET Framework do not all provide wrappers for safe performance under multiple threads
  - Using an enumerator is not thread-safe
    - Other threads may change the collection
    - `foreach` statement is not thread-safe either
    - Use the `lock` keyword to prevent other threads from using the collection
    - Use a `try` statement to catch the `InvalidOperationException`

COLLECTION INTERFACES

- All collection classes in the .NET Framework implement some combination of the collection interfaces.

Interface	Description
`ICollection`	The root 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.
`IEnumerable`	An object that can be enumerated. This interface contains exactly one method, GetEnumerator, which returns an IEnumerator object. ICollection implements IEnumerable, so all collection classes implement IEnumerable directly or indirectly.
`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.

GENERIC COLLECTION INTERFACES

Interface	Description
`ICollection(T)`	Defines methods to manipulate generic collections.
`IList(T)`	Represents a collection of objects that can be individually accessed by index.
`IEnumerable(T)`	Exposes the enumerator, which supports a simple iteration over a collection of a specified type.
`IEnumerator(T)`	Supports a simple iteration over a generic collection.
`IDictionary(TKey,TValue)`	Represents a generic collection of key/value pairs.
`IComparer(T)`	Defines a method that a type implements to compare two objects.