برنامه نویسی پیشرفته
C#

۱۲ آذر ۹۸

ملکی مجد

Topics

this session is based on chapter 17 of Microsoft Visual C# Step by Step, 8th Edition

- Recall generic
- Generic class
- Collections
  - List<T>

```
class Queue<T>{
	...
	private T[] data; // array is of type 'T' where 'T' is the type parameter
	public Queue(){
		this.data = new T[DEFAULTQUEUESIZE]; // use 'T' as the data type
	}
	public Queue(int size){
		...
		this.data = new T[size];
	}
	public void Enqueue(T item){ // use 'T' as the type of the method parameter
		...
	}
	public T Dequeue() {// use 'T' as the type of the return value
		T queueItem = this.data[this.tail]; // the data in the array is of type 'T'
		return queueItem;
	}
}
```

Queue of specific type

```
Queue<int> intQueue = new Queue<int>();
Queue<Horse> horseQueue = new Queue<Horse>();

intQueue.Enqueue(99);
int myInt = intQueue.Dequeue(); // no casting necessary
Horse myHorse = intQueue.Dequeue();
        // compiler error: cannot implicitly convert type 'int' to 'Horse'
```

Add *Comparability*

IComparable<T> interface (int CompareTo(T other);)

```
class Circle : System.IComparable<Circle>
{
    ...
    public int CompareTo(Circle other)
    {
        if (this.Area() == other.Area())
            return 0;

        if (this.Area() > other.Area())
            return 1;

        return -1;
    }
}
```

Creating a generic method

- specify the types of the parameters and the return type by using a type parameter
- used in conjunction with generic classes

the generic *Swap<T>* method

functionality is useful regardless of the type of data being swapped

```
static void Swap<T>(ref T first, ref T second){
        T temp = first;
        first = second;
        second = temp;
}

int a = 1, b = 2;
Swap<int>(ref a, ref b);
...
string s1 = "Hello", s2 = "World";
Swap<string>(ref s1, ref s2);
```

a convenient way to add a large number of items instead of repeated calls to the *Insert* method

```
static void InsertIntoTree<TItem>(ref Tree<TItem> tree,params TItem[] data)
                where TItem : IComparable<TItem>
{
        foreach (TItem datum in data) {
                if (tree == null)
                        {tree = new Tree<TItem>(datum);}
                else
                        { tree.Insert(datum);}
        }
}
```

Use InsertIntoTree<TItem

```
static void Main(string[] args)
{
    Tree<char> charTree = null;
    InsertIntoTree<char>(ref charTree, 'M', 'X', 'A', 'M', 'Z', 'Z', 'N');
    string sortedData = charTree.WalkTree();
    Console.WriteLine($"Sorted data is: {sortedData}");
}
```

Output: A M M N X Z Z

Collections

List<T>

Queue<T>

Stack<T>

LinkedList<T>

HashSet<T>

Dictionary<TKey, TValue>

SortedList<TKey, TValue>

- collections are generic types
- is optimized for a particular form of data storage and access
- provides specialized methods that support this functionality

The *List<T>* collection class

- an existing element in a *List<T>* collection by using **ordinary array notation**, with square brackets and the index of the element
- restrictions of an ordinary array
    - resize an array
    - remove an element from an array
    - insert an element into an array

Features of *List<T>*

- You don't need to specify the capacity of a *List<T>* collection when you create it; it **can grow and shrink as you add elements**. There is an overhead associated with this **dynamic** behavior, and if necessary you can specify an initial size. However, if you exceed this size, the *List<T>* collection simply grows as necessary.
- You can **remove a specified element** from a *List<T>* collection by using the ***Remove*** method. The *List<T>* collection automatically reorders its elements and closes the gap. You can also remove an item at a specified position in a *List<T>* collection by using the ***RemoveAt*** method.
- You can add an element to the end of a *List<T>* collection by using its ***Add*** method. You supply the element to be added. The *List<T>* collection resizes itself automatically.
- You can insert an element into the middle of a *List<T>* collection by using the ***Insert*** method. Again, the *List<T>* collection resizes itself.
- You can easily sort the data in a *List<T>* object by calling the ***Sort*** method.

Sample code

- Use a list<T>

- Note1: add *using System.Collections.Generic;*
- Note2: The way you determine the number of elements for a *List<T>* collection is different from querying the number of items in an array. When using a *List<T>* collection, you examine the ***Count*** property; when using an array, you examine the *Length* property.

The *Dictionary<TKey, TValue>* collection class

- The array and *List<T>* types provide a way to map an integer index to an element.
- ***associative array*** **:** sometimes you might want to implement a mapping in which the type from which you map is not an *int* but some other type, such as *string*

A *Dictionary<TKey, TValue>* collection cannot contain duplicate keys.

- If you call the *Add* method to add a **key that is already present** in the keys array, you'll get an **exception**. You can, however, use the square brackets notation to add a key/value pair (as shown in the following example) without danger of an exception, even if the key has already been added; any existing value with the same key will be **overwritten** by the new value. You can test whether a *Dictionary<TKey, TValue>* collection already contains a particular key by using the *ContainsKey* method.

Sample code

```
Dictionary<string, int> ages = new Dictionary<string, int>();
// fill the Dictionary
ages.Add("John", 51); // using the Add method
ages.Add("Diana", 50);
ages["James"] = 23; // using array notation
ages["Francesca"] = 21;
// iterate using a foreach statement // the iterator generates a KeyValuePair item
foreach (KeyValuePair<string, int> element in ages){
	string name = element.Key;
	int age = element.Value;
Console.WriteLine($"Name: {name}, Age: {age}");}
```

```
Output:
Name: John, Age: 51
Name: Diana, Age: 50
Name: James, Age: 23
Name: Francesca, Age: 21
```

Using collection initializers

- List<int> numbers = new List<int>(){10, 9, 8, 7, 7, 6, 5, 10, 4, 3, 2, 1};
- the C# compiler converts this initialization to a series of calls to the *Add* method
  - collections has to support the *Add* method
  - The *Stack<T>* and *Queue<T>* classes do not

```
Dictionary<string, int> ages = new Dictionary<string, int>(){
                    ["John"] = 51,
                    ["Diana"] = 50,
                    ["James"] = 23,
                    ["Francesca"] = 21};

Dictionary<string, int> ages = new Dictionary<string, int>(){
          {"John", 51},
          {"Diana", 50},
          {"James", 23},
          {"Francesca", 21}};
```

Find method in collection

- the argument to the *Find* method is a **predicate** that specifies the search criteria to use.

  The form of a predicate is a method that **examines each item in the collection** and returns a **Boolean** value indicating whether the item matches.

- In the case of the *Find* method, as soon as the first match is found, the corresponding item is returned.

lambda expression

- The easiest way to specify the predicate is to use a *lambda expression*.
- A lambda expression is an expression that returns a method.

New syntax! But do not worry!

- Lambda expressions do not define a method name, and the return type (if any) is inferred from the context in which the lambda expression is used.
- In the case of the *Find* method, the **predicate** processes **each item in the collection in turn**; the body of the predicate must **examine** the item and **return true or false** depending on whether it matches the search criteria

Sample code

The *Find* method returns the first item in the list that has the *ID* property set to 3:

```
struct Person{
public int ID { get; set; }
public string Name { get; set; }
public int Age { get; set; }}
...
// Create and populate the personnel list
List<Person> personnel = new List<Person>(){
new Person() { ID = 1, Name = "John", Age = 51 },
new Person() { ID = 2, Name = "Sid", Age = 28 },
new Person() { ID = 3, Name = "Fred", Age = 34 },
new Person() { ID = 4, Name = "Paul", Age = 22 },};
…
// Find the member of the list that has an ID of 3
Person match = personnel.Find((Person p) => { return p.ID == 3; });
//simplified  form:  Person match = personnel.Find(p => p.ID == 3);
```

the argument *(Person p) => { return p.ID == 3; }* is a lambda expression

- **A list of parameters enclosed in parentheses**. As with a regular method, if the method you are defining (as in the preceding example) takes no parameters, you must still provide the parentheses. In the case of the *Find* method, the predicate is provided with each item from the collection in turn, and this item is passed as the parameter to the lambda expression.
- The ***=>* operator**, which indicates to the C# compiler that this is a lambda expression.
- The **body of the method**. The example shown here is very simple, containing a single statement that returns a Boolean value indicating whether the item specified in the parameter matches the search criteria. However, **a lambda expression can contain multiple statements**, and you can format it in whatever way you feel is most readable. Just remember to add a semicolon after each statement, as you would in an ordinary method.

some examples of lambda expressions

x => x * x

// A simple expression that returns the square of its parameter The type of parameter x is inferred from the context.

x => { return x * x ; }

// Semantically the same as the preceding expression, but using a C# statement block as // a body rather than a simple expression

(int x) => x / 2

// A simple expression that returns the value of the parameter divided by 2 The type of parameter x is stated explicitly.

() => folder.StopFolding(0)

// Calling a method The expression takes no parameters. // The expression might or might not return a value.

(x, y) => { x++; return x / y; }

// Multiple parameters; the compiler infers the parameter types. The parameter x is passed by value, so the effect of the ++ operation is local to the expression.

(ref int x, int y) => { x++; return x / y; }

// Multiple parameters with explicit types Parameter x is passed by reference, so the effect of the ++ operation is permanent

features of lambda expressions

- If a lambda expression takes parameters, you specify them in the parentheses to the left of the => operator. You can omit the types of parameters, and the C# compiler will infer their types from the context of the lambda expression. You can pass parameters by reference (by using the *ref* keyword) if you want the lambda expression to be able to change its values other than locally, but this is not recommended.
- Lambda expressions can return values, but the return type must match that of the corresponding delegate.
- The body of a lambda expression can be a simple expression or a block of C# code made up of multiple statements, method calls, variable definitions, and other code items.
- Variables defined in a lambda expression method go out of scope when the method finishes.
- A lambda expression can access and modify all variables outside the lambda expression that are in scope when the lambda expression is defined. Be very careful with this feature!

Comparing arrays and collections

- A collection can dynamically resize itself as required.
  - An array instance has a fixed size
- A collection is linear
  - An array can have more than one dimension
- Not all collections support using an index
  - You store and retrieve an item in an array by using an index.
- Many of the collection classes provide a *ToArray* method that creates and populates an array containing the items in the collection.
  - Additionally, these collections provide constructors that can populate a collection directly from an array.