برنامه نویسی پیشرفته
C#

۱۰ آذر ۹۸

ملکی مجد

Topics

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

- Implement queue
- Generics
- Class generic Tree

model a first-in, first-out structure such as a **queue**

- class Queue
  - Fields
  - Constructor
  - Method Enqueue
  - Method Dequeue

```
class Queue{
        …
        public Queue() {…}
        public Queue(int size) {…}
        public void Enqueue(int item) {…}
        public int Dequeue() {…}
}
```

model a first-in, first-out structure such as a **queue**

- class Queue
  - Fields
  - Constructor
  - Method Enqueue
  - Method Dequeue

```
private const int DEFAULTQUEUESIZE = 100;
private int[] data;
private int head = 0, tail = 0;
private int numElements = 0;
```

model a first-in, first-out structure such as a **queue**

- class Queue
    - Fields
    - Constructor
    - Method Enqueue
    - Method Dequeue

```
public Queue(){
            this.data = new int[DEFAULTQUEUESIZE];
}
public Queue(int size){
            if (size > 0){
                        this.data = new int[size];      }
            else {
                        throw new ArgumentOutOfRangeException("size","Must be greater than zero");         }
}
```

model a first-in, first-out structure such as a **queue**

- class Queue
    - Fields
    - Constructor
    - Method Enqueue
    - Method Dequeue

```
public void Enqueue(int item){
        if (this.numElements == this.data.Length){
                throw new Exception("Queue full");}
        this.data[this.head] = item;
        this.head++;
        this.head %= this.data.Length;
        this.numElements++;
}
```

model a first-in, first-out structure such as a **queue**

- class Queue
    - Fields
    - Constructor
    - Method Enqueue
    - Method Dequeue

```
public int Dequeue(){
        if (this.numElements == 0){
                throw new Exception("Queue empty");}
        int queueItem = this.data[this.tail];
        this.tail++;
        this.tail %= this.data.Length;
        this.numElements--;
        return queueItem;
}
```

Using queue of int

```
Queue queue = new Queue(); // Create a new Queue
queue.Enqueue(100);
queue.Enqueue(-25);
queue.Enqueue(33);
Console.WriteLine($"{queue.Dequeue()}"); // Displays 100
Console.WriteLine($"{queue.Dequeue()}"); // Displays -25
Console.WriteLine($"{queue.Dequeue()}"); // Displays 33
```

Queue of other types

the *Queue* class works well for queues of *ints*,

but what if you want to **create queues of strings**, or floats, or even queues of more complex types such as *Circle ?*

One way around this restriction is to specify that the array in the *Queue* class contains items of type ***object***

Queue of object

update the constructors, and modify the *Enqueue* and *Dequeue* methods to take an *object* parameter and return an *object*

```
class Queue{
        ...
        private object[] data;
        public Queue(){
                this.data = new object[DEFAULTQUEUESIZE];}
        public Queue(int size){
                ...
                this.data = new object[size];}
        public void Enqueue(object item){
                ...}
        public object Dequeue(){
                ...
                object queueItem = this.data[this.tail];
                return queueItem;}
}
```

Use queue of object

```
Queue queue = new Queue();
Horse myHorse = new Horse();
queue.Enqueue(myHorse); // Now legal – Horse is an object
...
Horse dequeuedHorse = (Horse)queue.Dequeue();
                // Need to cast object back to a Horse
```

easy to write code with run-time error

```
Queue queue = new Queue();
Horse myHorse = new Horse();
queue.Enqueue(myHorse);
...
Circle myCircle = (Circle)queue.Dequeue(); // run-time error
```

throws a *System.InvalidCastException* exception at run time.

Another disadvantage of using the *object* approach

- consume additional memory and processor time if the runtime needs to convert an *object* to a value type and back again

```
Queue queue = new Queue();
int myInt = 99;
queue.Enqueue(myInt); // box the int to an object
...
myInt = (int)queue.Dequeue(); // unbox the object to an int
```

- Although boxing and unboxing happen transparently, they add **performance overhead** because they involve dynamic memory allocations.
- This overhead is small for each item, but it **adds up** when a program creates queues of large numbers of value types.

The generics solution

- C# provides **generics** to remove the need for casting, improve type safety, reduce the amount of boxing required, and make it easier to create generalized classes and methods.

- Generic classes and methods **accept *type parameters*,**

  which specify the types of objects on which they operate

generic class

- you indicate that a class is a generic class by providing a **type parameter** in angle brackets

```
class Queue<T>
{
        ...
}
```

The *T* in this example acts as a **placeholder** for a **real type** at **compile time**

```
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>();
```

Now:

compiler has enough information to perform strict type checking

no longer need to cast data when you call the *Dequeue* method

compiler can trap any type mismatch errors early

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

```
struct Person{...}
…
Queue<int> intQueue = new Queue<int>();
Queue<Person> personQueue = new Queue<Person>();
```

compiler also generates the versions of the *Enqueue* and *Dequeue* methods for each queue : does **not require boxing or unboxing**

```
public void Enqueue(int item);public int Dequeue();
public void Enqueue(Person item);public Person Dequeue();
```

Type parameter

- The type parameter does not have to be a simple class or value type

```
Queue<Queue<int>> queueQueue = new Queue<Queue<int>>();
```

note

- A generic class can have multiple type parameters
- You can also define **generic structures and interfaces** by using the same type-parameter syntax as for generic classes

Generics vs. generalized classes

- **Generalized**

  generalized class designed to take parameters that can be cast to different types.

  There is a *single* implementation of this class, and its methods take *object* parameters and return *object* types

- **Generics**

  Queue<T> class

  You can think of a generic class as one that defines a template that is then used by the compiler to generate new type-specific classes on demand

  Queue<int> and Queue<Horse> : distinctly different types

Generics and constraints

- limit the type parameters of a generic class

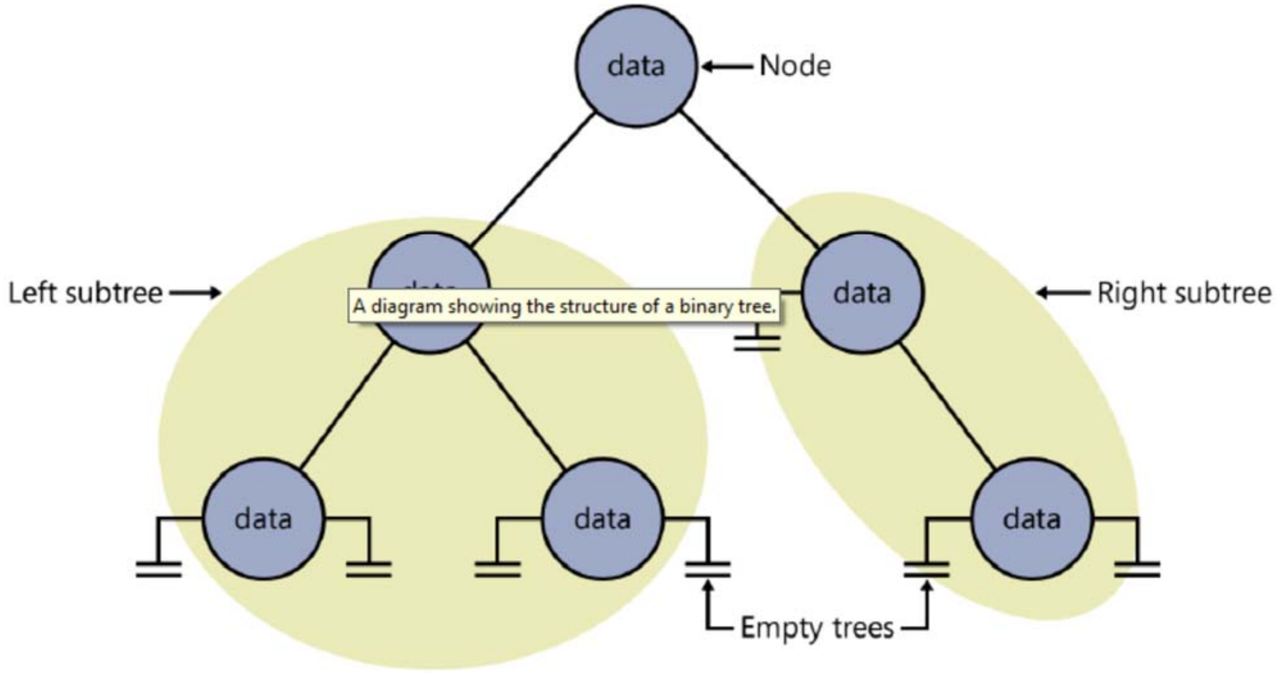
public class PrintableCollection<T> **where T : Iprintable**

the compiler checks to be sure that the type used for *T* actually implements the *IPrintable* interface; if it doesn't, it stops with a compilation error.

Example
Creating a generic class

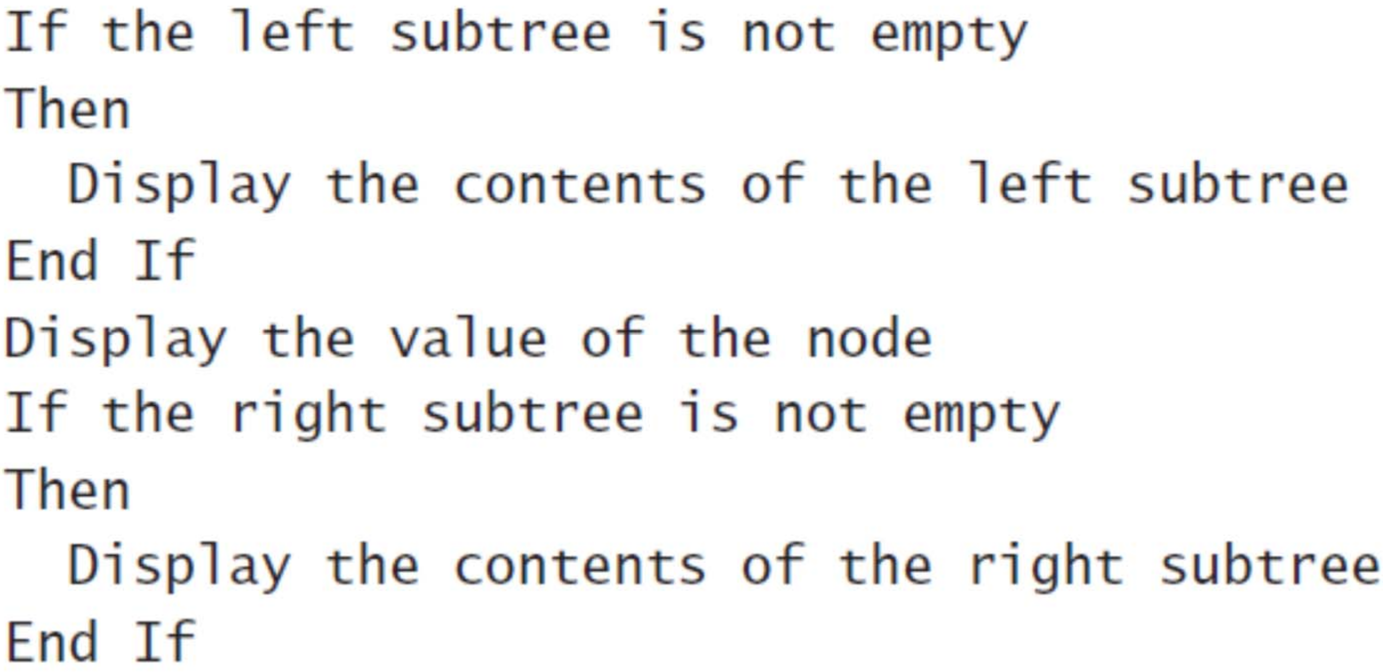
- Binary tree
- A binary tree is a **recursive** (self-referencing) data structure that can be **empty** or contain **three elements**: a datum, which is typically referred to as the *node*, and two subtrees, which are themselves binary trees.

- The real power of binary trees becomes evident when you use them **for sorting data**. If you start with an unordered sequence of objects of the same type, you can **construct an ordered** binary tree and then **walk through** the tree to visit each node in an ordered sequence


data
Node
Left subtree
data
A diagram showing the structure of a binary tree.
data
Right subtree
data
data
data
Empty trees

```
If the tree, B, is empty
Then
  Construct a new tree B with the new item I as the node, and empty left and
  right subtrees
Else
  Examine the value of the current node, N, of the tree, B
  If the value of N is greater than that of the new item, I
  Then
    If the left subtree of B is empty
    Then
      Construct a new left subtree of B with the item I as the node, and
      empty left and right subtrees
    Else
      Insert I into the left subtree of B
    End If
  Else
    If the right subtree of B is empty
    Then
      Construct a new right subtree of B with the item I as the node, and
      empty left and right subtrees
    Else
      Insert I into the right subtree of B
    End If
  End If
End If
```

Walk through tree



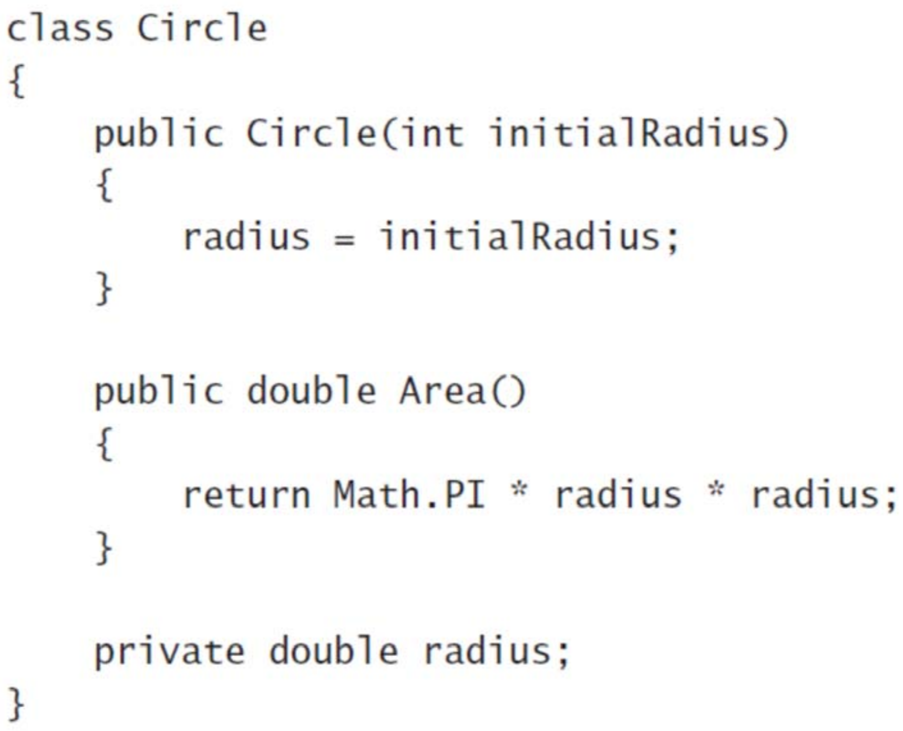
```
If the left subtree is not empty
Then
  Display the contents of the left subtree
End If
Display the value of the node
If the right subtree is not empty
Then
  Display the contents of the right subtree
End If
```

The *System.IComparable* and *System.IComparable<T>* interfaces

- The algorithm for inserting a node into a binary tree requires you to compare the value of the node that you are inserting with nodes already in the tree.

Add *Comparability*

before



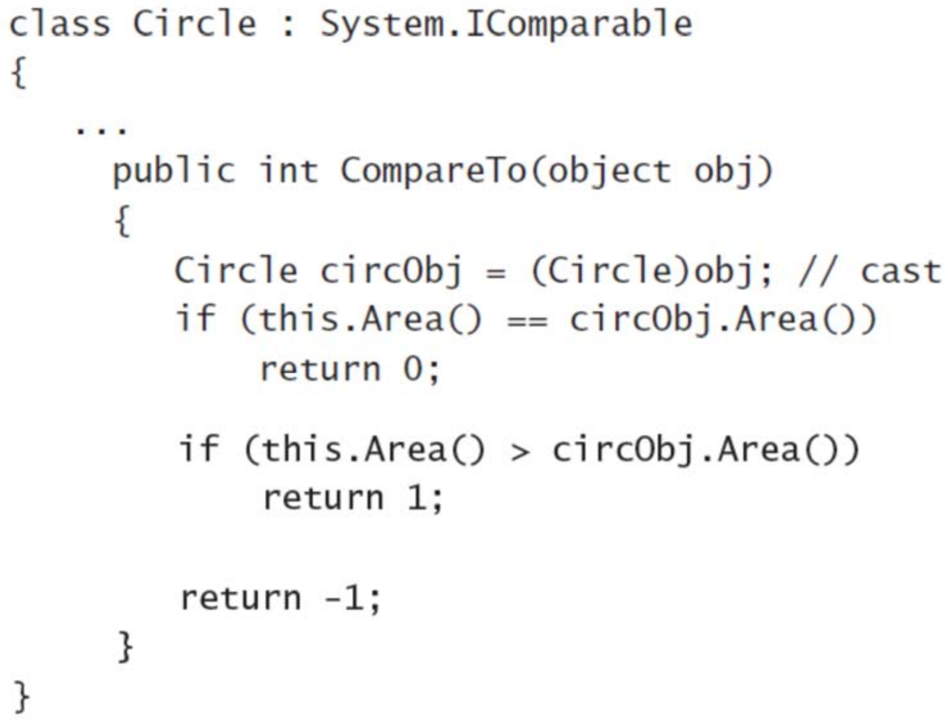
```
class Circle
{
    public Circle(int initialRadius)
    {
        radius = initialRadius;
    }

    public double Area()
    {
        return Math.PI * radius * radius;
    }

    private double radius;
}
```

Add *Comparability*

System.IComparable



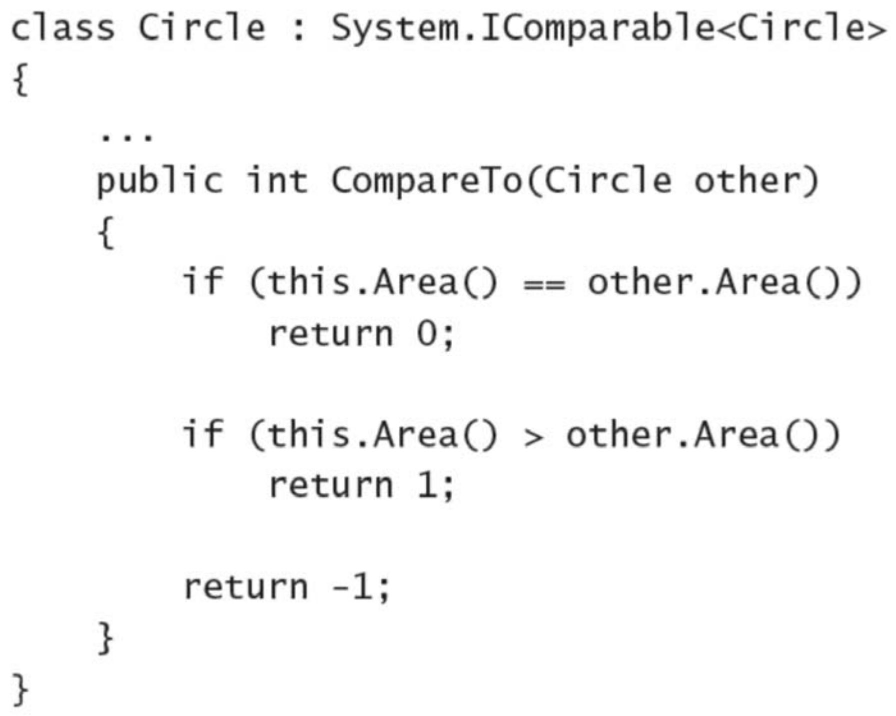
```
class Circle : System.IComparable
{
   ...
     public int CompareTo(object obj)
     {
        Circle circObj = (Circle)obj; // cast
        if (this.Area() == circObj.Area())
            return 0;

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

        return -1;
     }
}
```

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;
    }
}
```