برنامه نویسی پیشرفته
C#

۲۱ آبان ۹۸

ملکی مجد

topics

- Using garbage collection and resource management

- Review
- The life and times of an object
- Writing destructors
- Why use the garbage collector?

review

- how to create variables and objects
- how memory is allocated when you create variables and objects ?
  - stack
  - Heap
- Life time
  - Value type
  - Reference type

The life and times of an object

```
int sizeOfSquare = 99;
Square mySquare = new Square(sizeOfSquare); //Square is a reference type
```

object creation is really a two-phase process:

1. The *new* operation **allocates a chunk of *raw* memory** from the heap. You have no control over this phase of an object's creation.
2. The *new* operation **converts** the chunk of raw memory to an object; it has **to initialize the object**. You can **control** this phase by using a **constructor**.

- can access the members of an object by using the dot operator (.).
  - mySquare.Draw();

- When the *mySquare* variable goes out of scope
  - the *Square* object is no longer being actively referenced.
- The object can then be destroyed, and the memory that it is using can be reclaimed

- object destruction is a two-phase process

1. The common language runtime (CLR) must perform some tidying up. You can control this by writing a *destructor*.
2. The CLR must return the memory previously belonging to the object back to the heap; the memory that the object lived in must be deallocated. You have no control over this phase.

garbage collection

- The process of destroying an object and returning memory back to the heap is known as *garbage collection*.

Destructors

- You can use a destructor to **perform any tidying up** that's required when an object is garbage collected.
- The CLR will **automatically clear up** any managed resources that an object uses, so in many of these cases, writing a destructor is **unnecessary** .

- if a resource is large
  - a destructor can prove useful.

Writing destructors

- A destructor is a special method, a little like a constructor, except that the CLR calls it **after the reference to an object has disappeared**

- a tilde (~) followed by the name of the class.

example

```
class FileProcessor{
        FileStream file = null;
        public FileProcessor(string fileName) {
                this.file = File.OpenRead(fileName); //open file for reading
        }
        ~FileProcessor() {
        this.file.Close(); // close file
        }
}
```

Restrictions

- Destructors apply only to reference types; you cannot declare a destructor in a value type, such as a *struct*

- You cannot specify an access modifier (such as *public*) for a destructor.

- A destructor cannot take any parameters.

Internally, the C# compiler automatically translates a destructor into an override of the *Object.Finalize* method.

```
class FileProcessor
{
        ~FileProcessor()
        {
            //your code goes here
        }
}
```

```
class FileProcessor
{
        protected override void Finalize()
        {
                try {
                    //your code goes here
                }
                finally{
                     base.Finalize();
                }
        }
}
```

- only the compiler can make this translation. You can’t write your own method to override *Finalize*, and you can’t call *Finalize* yourself

Why use the garbage collector?

- You can never destroy an object yourself by using C# code. There just isn’t any syntax to do it.

- How many references can you create to an object?
  - managing object lifetimes is complex
  - C# decided to prevent your code from taking on this responsibility

The garbage collector makes the following guarantees:

- Every object will be destroyed, and its destructor will be run. When a program ends, all outstanding objects will be destroyed.
- Every object will be destroyed exactly once.
- Every object will be destroyed only when it becomes unreachable—that is, when there are no references to the object in the process running your application.

- you should never write code that depends on destructors running in a particular sequence or at a specific point in your application.

garbage collector thread

- A ***thread*** is a separate path of execution in an application. Windows uses threads to enable an application to perform multiple operations concurrently.

- The garbage collector runs in its **own thread** and can execute only at certain times

How does the garbage collector work?(at a high level)

1. It builds a map of all reachable objects
2. It checks whether any of the unreachable objects has a destructor that needs to be run (a process called *finalization*)(is placed in a special queue called the *freachable queue*)
3. It deallocates the remaining unreachable objects (those that don't require finalization) by moving the *reachable* objects down the heap
4. At this point, it allows other threads to resume.
5. It finalizes the unreachable objects that require finalization (now in the *freachable* queue) by running the *Finalize* methods on its own thread.

Resource management

- some resources are just too valuable to lie around waiting for an arbitrary length of time until the garbage collector actually releases them
  - memory, database connections, or file handles

- release the resource yourself.
  - create a *disposal method*—a method that explicitly disposes of a resource.

example

```
TextReader reader = new StreamReader(filename);
string line;
while ((line = reader.ReadLine()) != null)
{
    Console.WriteLine(line);
}
reader.Close();
```

```
try
{
    string line;
    while ((line = reader.ReadLine()) != null)
    {
        Console.WriteLine(line);
    }
}
finally
{
      reader.Close();
}
```

Drawbacks of finally

- It quickly becomes unwieldy if you have to dispose of more than one resource. (You end up with nested *try* and *finally* blocks.)
- In some cases, you might need to modify the code to make it fit this idiom. (For example, you might need to reorder the declaration of the resource reference, remember to initialize the reference to *null*, and remember to check that the reference isn't *null* in the *finally* block.)
- It fails to create an abstraction of the solution. This means that the solution is hard to understand and you must repeat the code everywhere you need this functionality.
- The reference to the resource remains in scope after the *finally* block. This means that you can accidentally try to use the resource after it has been released

using statement

- The *using* statement provides a clean mechanism for controlling the lifetimes of resources. You can create an object, and this object will be destroyed when the *using* statement block finishes.

- Syntax:

```
using ( type variable = initialization )
{
        StatementBlock
}
```

example

```
using (TextReader reader = new StreamReader(filename))
{
    string line;
    while ((line = reader.ReadLine()) != null)
    {
        Console.WriteLine(line);
    }
}
```

Equivalent code!

```
{
    TextReader reader = new StreamReader(filename);
    try
    {
       string line;
       while ((line = reader.ReadLine()) != null)
       {
           Console.WriteLine(line);
       }
    }
    finally
    {
         if (reader != null)
         {
            ((IDisposable)reader).Dispose();
         }
    }
}
```

interface IDisposable

- The variable you declare in a *using* statement must be of a type that implements the *IDisposable* interface. The *IDisposable* interface lives in the *System* namespace and contains just one method, named *Dispose*:

```
namespace System
{
        interface IDisposable
        {
                void Dispose();
        }
}
```

- the *StreamReader* class implements the *IDisposable* interface, and its *Dispose* method calls *Close* to close the stream