**Garbage Collection and Destructors:**

* As you have seen, objects are dynamically allocated from a pool of free memory by using

the **new** operator.

* Of course, memory is not infinite, and the free memory can be exhausted.
* Thus, it is possible for **new** to fail because there is insufficient free memory to create the

desired object.

* For this reason, one of the key components of any dynamic allocation scheme is the recovery of free memory from unused objects, making that memory available for subsequent reallocation.
* In many programming languages, the release of previously allocated memory is handled manually.
* For example, in C++, the **delete** operator is used to free memory that was allocated.
* However, C# uses a different, more trouble-free approach: *garbage collection.*
* C#’s garbage collection system reclaims objects automatically—occurring transparently, behind the scenes, without any programmer intervention.
* It works like this: When no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object is eventually released and collected. This recycled memory can then be used for a subsequent allocation.
* Garbage collection occurs only sporadically during the execution of your program.
* It will not occur simply because one or more objects exist that are no longer used.
* Thus, you can’t know, or make assumptions about, precisely when garbage collection will take place.

**Destructors:**

* It is possible to define a method that will be called just prior to an object’s final destruction by the garbage collector.
* This method is called a *destructor,* and it can be used in some highly specialized situations to ensure that an object terminates cleanly.
* For example, you might use a destructor to ensure that a system resource owned by an object is released. It must be stated at the outset that destructors are a very advanced feature that are applicable only to certain rare cases.
* They are not normally needed. They are briefly described here for completeness.
* Destructors have this general form:

~*class-name*( )

{

// destruction code

}

* Here, *class*-*name* is the name of the class. Thus, a destructor is declared like a constructor except that it is preceded with a ~ (tilde). Notice it has no return type and takes no arguments.
* To add a destructor to a class, you simply include it as a member.
* It is called whenever an object of its class is about to be recycled.
* Inside the destructor, you will specify those actions that must be performed before an object is destroyed.
* It is important to understand that the destructor is called just prior to garbage collection.
* It is not called when a variable containing a reference to an object goes out of scope, for example. (This differs from destructors in C++, which *are* called when an object goes out of scope.) This means that you cannot know precisely when a destructor will be executed.
* Furthermore, it is possible for your program to end before garbage collection occurs, so a destructor might not get called at all.
* The following program demonstrates a destructor. It works by creating and destroying a large number of objects. During this process, at some point the garbage collector will be activated, and the destructors for the objects will be called.

// Demonstrate a destructor.

using System;

class Destruct

{

public int x;

public Destruct(int i)

{

x = i;

}

// Called when object is recycled.

~Destruct()

{

Console.WriteLine("Destructing " + x);

}

// Generates an object that is immediately destroyed.

public void Generator(int i)

{

Destruct o = new Destruct(i);

}

}

class DestructDemo

{

public static void Main()

{

int count;

Destruct ob = new Destruct(0);

/\* Now, generate a large number of objects. At some point, garbage collection will occur. Note: You might need to increase the number of objects generated in order to force garbage collection. \*/

for(count=1; count < 100000; count++)

ob.Generator(count);

Console.WriteLine("Done");

}

}

* Here is how the program works.
* The constructor sets the instance variable **x** to a known value.
* In this example, **x** is used as an object ID. The destructor displays the value of **x** when an object is recycled. Of special interest is **Generator( )**.
* This method creates and then promptly destroys a **Destruct** object.
* The **DestructDemo** class creates an initial **Destruct** object called **ob**. Then using **ob**, it creates 100,000 objects by calling **Generator( )** on **ob**. This has the net effect of creating and destroying 100,000 objects. At various points in the middle of this process, garbage collection will take place.
* Precisely how often or when is dependent upon several factors, such as the initial amount of free memory, the operating system, and so on.
* However, at some point, you will start to see the messages generated by the destructor.
* If you don’t see the messages prior to program termination (that is, before you see the “Done”

message), try increasing the number of objects being generated by upping the count in the **for** loop.

* One important point: The call to **WriteLine( )** inside **~Destruct( )** is purely for the sake of illustration in this rather contrived example.
* Normally, a destructor should act only on the instance variables defined by its class.
* Because of the nondeterministic way in which destructors are called, they should not be used to perform actions that must occur at a specific point in your program. One other point: It is possible to request garbage collection.
* However, manually initiating garbage collection is not recommended for most circumstances, because it can lead to inefficiencies.
* Also, because of the way the garbage collector works, even if you explicitly request garbage collection, there is no way to know precisely when a specific object will be recycled.