Nondeterministic Destruction, Delegates, and Exceptions

Last Week

- We
  - Looked at interfaces and enumerations
  - Covered boxing and unboxing

Garbage Collection

- The basic idea: memory in the managed heap that is no longer needed is automatically reclaimed.
- The programmer is not responsible for freeing it.
- Object life cycle:
  - Allocate memory
  - Initialize the object to a useful state
  - Use the object
  - Tear the object down ← The garbage collector can't do this
  - Free memory ← The garbage collector is solely responsible for this

C# "Destructors" : the Finalize method

- **`System.Object`** has a protected virtual method **`Finalize`**
- Called when the garbage collector determines that the object is garbage but before the memory is reclaimed
- Types that require cleanup should override this method (e.g., to close files or network connections)
- In C#, the syntax is like the destructor syntax of C++:

```
~MyClass()
{ //cleanup here }
```

- The finalization function is never explicitly called, and destruction of an object in non-deterministic

Use Finalization only when necessary

- Finalizable objects take longer to allocate
- Forcing the garbage collector to call a Finalize method can hurt performance. If there is an array of 10,000 objects, the finalization method gets called 10,000 times
- You have no control over when finalization takes place
- The CLR doesn't guarantee the order in which finalization methods are called

Garbage Collection Example

```
class MyClass
{
 private int[] array;
 private Helper[] helpers;
 private int n;
 public static int numFinalized = 0;
 public static int numAlloc = 0;
 public MyClass(int num)
 {
  n = num;
  array = new int[n]; //array of n ints
  helpers = new Helper[n]; //array of n Helper refs
  numAlloc++;
  for(int i = 0; i < n; i++)
  {
   array[i] = i;
   helpers[i] = new Helper();
  }
```

Garbage Collection Example

```
~MyClass()
{
 numFinalized++;
 Console.WriteLine("Finalize called on MyClass" +
                   " {0}, {1} allocated, {2} finalized",
                   n, numAlloc, numFinalized);
}

public void Print()
{
 foreach (int i in array)
 Console.Write(i + " ");
 Console.WriteLine();
}
```

Garbage Collection Example

```
static void Main(string[] args)
{
 MyClass mc = new MyClass(101);
 mc = new MyClass(10);
 mc.Print();
 for (int i = 1; i <= 50; i++)
  mc = new MyClass(i * 100);
 Console.ReadLine();
 Console.WriteLine("Bye bye");
}
class Helper
{
 private int[] array = new int[100];
}
}
```

Delegates

- Simplest description: type-safe function pointers
- Declaring a delegate defines a class that derives from System.MulticastDelegate, which derives from System.Delegate

Delegates

- Delegates maintain a list of methods to be called when the delegate is invoked
- To insure type safety, the signature (return type and arguments) of acceptable callback methods is specified when the delegate class is defined
- We instantiate a delegate by "wrapping" the callback method.
- We can add methods to the callback chain or remove methods from the chain.

Delegates Example

```
class Worker
 {
   public void Advise(Boss boss) { aBoss = boss; }
   public void DoWork()
   {
     Console.WriteLine("Worker: work started");
     if( aBoss != null ) aBoss.WorkStarted();

     Console.WriteLine("Worker: work progressing");
     if( aBoss != null ) aBoss.WorkProgressing();

     Console.WriteLine("Worker: work completed");
     if( aBoss != null )
     {
       int grade = aBoss.WorkCompleted();
       Console.WriteLine("Worker grade = " + grade);
     }
   }
   private Boss aBoss;
```

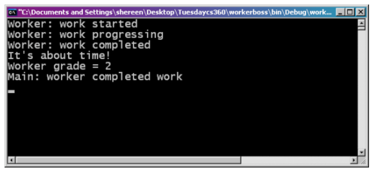
Delegates Example

```
class Boss
 {
   public void WorkStarted() {
           /* boss doesn't care. */ }
   public void WorkProgressing() {
           /* boss doesn't care. */ }
   public int WorkCompleted()
   {
     Console.WriteLine("It's about time!");
     return 2; /* out of 10 */
   }
 }
```

Delegates Example

```
class Universe
 {
   static void Main()
   {
     Worker peter = new Worker();
     Boss boss = new Boss();
     peter.Advise(boss);
     peter.DoWork();

     Console.WriteLine("Main: worker completed work");
     Console.ReadLine();
   }
 }
```

Output



"C:\Documents and Settings\shereen\Desktop\Tuesdaycs360\workerboss\bin\Debug\work...

```
Worker: work started
Worker: work progressing
Worker: work completed
It's about time!
Worker grade = 2
Main: worker completed work
```

Delegates Example

- What if the worker wanted to advise more than just their boss
- How can we split the following methods into an interface:
  - `WorkStarted`
  - `WorkProgressing`
  - `WorkCompleted`

Delegates Example

```
interface IWorkerEvents
  {
    void WorkStarted();
    void WorkProgressing();
    int WorkCompleted();
  }
```

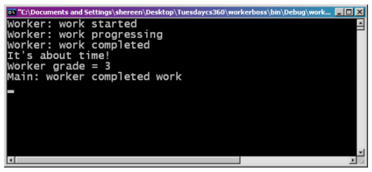
Delegates Example

```
class Worker
 {
   public void Advise(IWorkerEvents events) { theEvents = events; }
   public void DoWork()
   {
     Console.WriteLine("Worker: work started");
     if( theEvents != null ) theEvents.WorkStarted();

     Console.WriteLine("Worker: work progressing");
     if(theEvents != null ) theEvents.WorkProgressing();

     Console.WriteLine("Worker: work completed");
     if(theEvents != null )
     {
       int grade = theEvents.WorkCompleted();
       Console.WriteLine("Worker grade = " + grade);
     }
   }
   private IWorkerEvents theEvents;
 }
```

Delegates Example

```
class Boss : IWorkerEvents
 {
   public void WorkStarted() {
              /* boss doesn't care. */ }
   public void WorkProgressing() {
              /* boss doesn't care. */ }
   public int WorkCompleted()
   {
     Console.WriteLine("It's about time!");
     return 3; /* out of 10 */
   }
 }
```

Delegates Example


"C:\Documents and Settings\shereen\Desktop\Tuesdaycs360\workerboss\bin\Debug\work...

```
Worker: work started
Worker: work progressing
Worker: work completed
It's about time!
Worker grade = 3
Main: worker completed work
```

Delegates Example

- Still, his boss complained bitterly. "Peter!" his boss fumed. "Why are you bothering to notify me when you start your work or when your work is progressing?!? I don't care about those events. Not only do you force me to implement those methods, but you're wasting valuable work time waiting for me to return from the event, which is further expanded when I am far away! Can't you figure out a way to stop bothering me?"

Delegates Example

- Interfaces are not suitable for events
- Break methods out of interface into separate delegate functions

Delegates Example

```
delegate void WorkStarted();

delegate void WorkProgressing();

delegate int WorkCompleted()
```

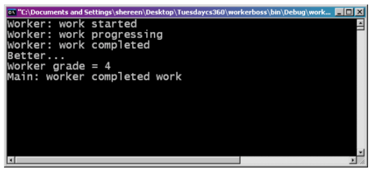
Delegates Example

```
class Worker
 {
   public void DoWork()
   {
     Console.WriteLine("Worker: work started");
     if( started != null ) started();

     Console.WriteLine("Worker: work progressing");
     if( progressing != null ) progressing();

     Console.WriteLine("Worker: work completed");
     if( completed != null )
     {
       int grade = completed();
       Console.WriteLine("Worker grade = " + grade);
     }
   }
   public WorkStarted started;
   public WorkProgressing progressing;
   public WorkCompleted completed;
 }
```

Delegates Example

```
class Boss
 {
   public int WorkCompleted()
   {
     Console.WriteLine("Better...");
     return 4; /* out of 10 */
   }
 }
 class Universe
 {
   static void Main()
   {
     Worker peter = new Worker();
     Boss boss = new Boss();
     peter.completed = new WorkCompleted(boss.WorkCompleted);
     peter.DoWork();

     Console.WriteLine("Main: worker completed work");
     Console.ReadLine();
   }
 }
```

Delegates Example


"C:\Documents and Settings\shereen\Desktop\Tuesdaycs360\workerboss\bin\Debug\work...

```
Worker: work started
Worker: work progressing
Worker: work completed
Better...
Worker grade = 4
Main: worker completed work
```

Exceptions

```
public void SomeMethod(...)
{
File file = new File("Readme.txt");
...
file.Close();
}
```

What happens if the file doesn't exist?

What happens if an error occurs in here?

Exceptions: the general idea

```
try
{
 some code that might throw an exception
 more code
}
catch (most specific exception)
{
 handle the exception
}
catch (less specific exception)
{
 handle the exception
}
catch (any exception)
{
 handle the exception
}
finally
{
 do this no matter what
}
still more code
```

Exceptions

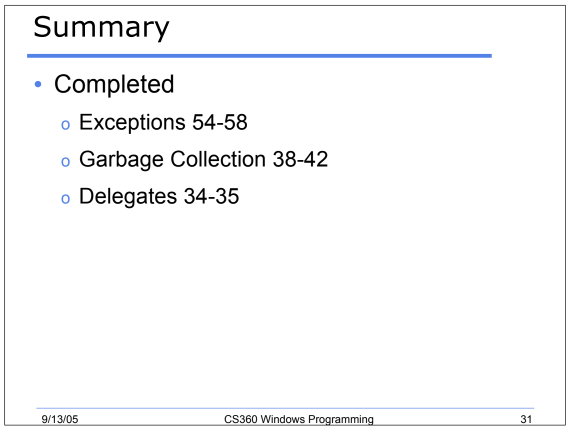
```
using System.IO;
public void SomeMethod(...)
{
 File file = null;
 try
 {
  file = new File("Readme.txt");
  more code
 }
 catch (FileNotFoundException e)
 {
  Console.WriteLine("File " + e.FileName + " not found");
 }
 catch (Exception e)
 {
  Console.WriteLine(e);
 }
 finally
 {
  if (file != null)
  file.Close();
 }
}
```

Exceptions

- In a catch block, you can:
  - Rethrow the same exception, notifying code higher in the call stack
  - Throw a different exception, giving additional information to code higher in the call stack
  - Handle the exception and fall out the bottom of the catch block

Exceptions

- Remember that:
  - Exceptions are not always "errors"
  - Exceptions are not always infrequent
  - Sometimes it's best not to catch an exception where it occurs
  - There is a performance hit for exceptions



Summary

- Completed
  - Exceptions 54-58
  - Garbage Collection 38-42
  - Delegates 34-35