Phi Consulting

.Net Training

Advanced Basics of .Net (C#)

© 2008 Phi Consulting Proprietary and Confidential

Φ Phi Consulting

Agenda

- Reflection Overview
- Attributes Overview
- Regular Expressions Overview
- Memory Management & Garbage Collection
- Helpful Namespaces and Objects

© 2008 Phi Consulting Proprietary and Confidential

Agenda

- Reflection Overview

Phi Consulting

Reflection Overview

- What is Reflection?
- Reflection provides objects (of type Type) that encapsulate assemblies, modules and types
- You can use reflection to dynamically
  - create an instance of a type, bind the type to an existing object
  - get the type from an existing object and invoke its methods or access its fields and properties
  - If you are using attributes in your code, Reflection enables you to access them

© 2008 Phi Consulting

Phi Consulting

Reflection Overview

- Reflection is useful in the following situations:
  - When you need to access attributes in your program's metadata
  - For examining and instantiating types in an assembly
  - For building new types at runtime. Use classes in System.Reflection.Emit
  - For performing late binding, accessing methods on types created at run time

Phi Consulting

Reflection Overview

- Reflection provides objects that encapsulate assemblies, modules, and types
- You can use reflection to dynamically create an instance of a type, bind the type to an existing object, or get the type from an existing object
- You can then invoke the type's methods or access its fields and properties

© 2008 Phi Consulting Proprietary and Confidential

Reflection Overview

- Use **Assembly** to define and load assemblies, load modules that are listed in the assembly manifest, and locate a type from this assembly and create an instance of it
- Use **Module** to discover information such as the assembly that contains the module and the classes in the module. You can also get all global methods or other specific, nonglobal methods defined on the module
- Use **PropertyInfo** to discover information such as the name, data type, declaring type, reflected type, and read-only or writable status of a property, and to get or set property values
- Use **MethodInfo** to discover information such as the name, return type, parameters, access modifiers (such as public or private), and implementation details (such as abstract or virtual) of a method. Use the GetMethods or GetMethod method of a Type to invoke a specific method
- Use **FieldInfo** to discover information such as the name, access modifiers (such as public or private) and implementation details (such as static) of a field, and to get or set field values

Reflection Overview

- Use **EventInfo** to discover information such as the name, event-handler data type, custom attributes, declaring type, and reflected type of an event, and to add or remove event handlers
- Use **ConstructorInfo** to discover information such as the name, parameters, access modifiers (such as public or private), and implementation details (such as abstract or virtual) of a constructor. Use the GetConstructors or GetConstructor method of a Type to invoke a specific constructor
- Use **ParameterInfo** to discover information such as a parameter's name, data type, whether a parameter is an input or output parameter, and the position of the parameter in a method signature
- Use **CustomAttributeData** to discover information about custom attributes when you are working in the reflection-only context of an application domain. CustomAttributeData allows you to examine attributes without creating instances of them

Phi Consulting

Reflection Overview

- Example:

```
// Using GetType to obtain type information:
int i = 42;
System.Type type = i.GetType();
MethodInfo[] methods = t.GetMethods();
foreach (MethodInfo method in methods)
{
    Console.WriteLine("Method: {0}", method.Name);
    Console.WriteLine("Parameters:");
    ParameterInfo[] parameters = method.GetParameters();
    foreach (ParameterInfo parameter in parameters)
    {
        Console.WriteLine("/t{0} ({1})", parameter.Name, parameter.ParameterType);
    }
}
Console.WriteLine(type.InvokeMethod("ToString");
```

Reflection Overview

- Example:

```
// Using Assembly to obtain type information:
 Assembly a;
 Type[] types = a.GetTypes();
foreach (Type t in types)
{
      Console.WriteLine(t.ToString());
}
//Late binding
string s = (string)a.CreateInstance("System.String");
```

Phi Consulting

Reflection Overview

- Reflection has a cost in performance penalty
- Some parts of Reflection cost more than others
- While getting Type information is cheap, CreateInstance() is very expensive, especially if called from Activator.CreateInstance()

Agenda

- Attributes Overview

© 2008 Phi Consulting Proprietary and Confidential

Phi Consulting

Attributes

- Attributes provide a powerful method of associating declarative information with C# code (types, methods, properties, and so forth)
- Attributes add metadata to your program. Metadata is information embedded in your program such as compiler instructions or descriptions of data
- Your program can examine its own metadata using Reflection
- You can construct code that extends behavior based on Attributes detected through Reflection
- Attributes are commonly used when interacting with COM

Phi Consulting

Attributes

- The syntax is simple – square bracket notation
- In this example, the attribute System.Reflection.TypeAttributes.Serializable is used to apply a specific characteristic to a class:

```
[System.Serializable]
public class SampleClass
{
   // Objects of this type can be serialized.
}
```

Phi Consulting

Attributes

- Attributes are commonly used in .Net Web Services – they take the work out of creating web services

```
[WebService(Namespace="HealthMarkets.com")]
public class MyWebService : WebService
{
    [WebMethod]
    public string HelloWorld()
    { return "Hello World!"; }
}
```

- This simple construct will generate all the XML, XSD schema and WSDL necessary to create a web service

Phi Consulting

Attributes

- You can construct your own Attributes or custom behavior based on Attributes
- Throughout the .Net Framework there are many Attributes that extend the basic functionality of your classes
- Just be aware of them when you see them

Phi Consulting

Accessing Attributes With Reflection

- Given an Attribute called Author:

```
[Author("H. Ackerman", version = 1.1)]
class SampleClass { … }

 private void PrintAuthorInfo(System.Type t)
   {
       System.Console.WriteLine("Author information for {0}", t);
       System.Attribute[] attrs = System.Attribute.GetCustomAttributes(t);  // reflection

       foreach (System.Attribute attr in attrs)
       {
           if (attr is Author)
           {
               Author a = (Author)attr;
               System.Console.WriteLine("   {0}, version {1:f}", a.GetName(), a.version);
           }
       }
   }
```

Phi Consulting

Agenda

- Regular Expressions Overview

© 2008 Phi Consulting Proprietary and Confidential

Φ Phi Consulting

Regular Expressions

- Regular Expressions are used for pattern matching
- Not an exclusive .Net concept – RegExp's have been around for many years
- .Net supports via the Regex class in the System.Text.RegularExpressions namespace
- Regular Expressions have their unique syntax
- There are THOUSANDS of pre-made Regular Expressions out there
- Google them before making your own

Phi Consulting

Regular Expressions

- Example – email matching pattern:

```
using System.Text.RegularExpressions;
Regex emailregex = new Regex("(?<user>[^@]+)@(?<host>.+)");

String s = "johndoe@tempuri.org";

Match m = emailregex.Match(s);
if ( m.Success ) {
        Console.WriteLine("User: " + m.Groups["user"].Value);
        Console.WriteLine("Host: " + m.Groups["host"].Value);
 } else {
        Console.WriteLine(s + " is not a valid email address");
}
Console.WriteLine();
```

- Use the Match method to pass variable and return a new Match object
- The Match object will return regardless of whether any matches were found
- The Success property tells you if a match was found

© 2008 Phi Consulting Proprietary and Confidential

Regular Expressions

- For more details on Regular Expression syntax:
  - http://www.regular-expressions.info
- All about syntax and tutorials on how to use RegEx's in different languages

Phi Consulting

Agenda

- Memory Management & Garbage Collection
  - GC Generations Overview
  - Finalization (Destructors/Dispose())

Phi Consulting

Garbage Collector

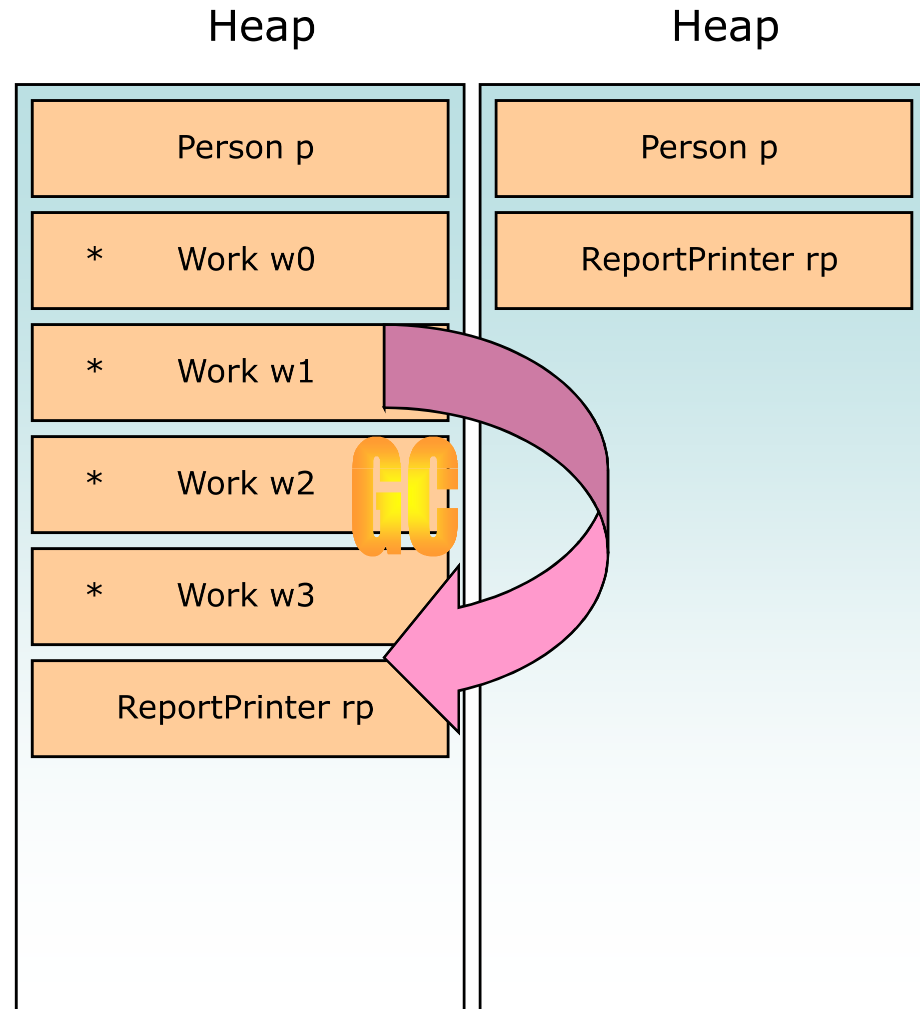
- In traditional C/C++ programming you have to allocate memory using a function such as malloc() and then be responsible for deallocating that memory yourself
- If you don't you have a memory leak
- Also, malloc() makes memory allocation expensive
- In .Net a Garbage Collector system was designed, instead
- Garbage Collector systems trade simplicity at allocation time for complexity at cleanup time
- Custom allocators are a lot more effective and fast, but they are also a lot of work to write, understand and maintan

Garbage Collector

- What is the Garbage Collector?
- In simple terms it is simply an overseer of the managed heap
- Objects get allocated to the heap as they are created, in the order they are created
- Periodically the GC will check for UNREACHABLE objects and reclaim their memory
- The GC then compacts the used memory, so free memory is defragmented
- Objects that survive collection get to stick around a while longer
- (To improve performance, large objects (>20K) are allocated from a large object heap)
- The GC decides WHEN to collect – usually when there's idle time

Mark and Compact

```
public static void Main()
{
   Person p = new Person("John", "Smith");
   for(int i=0; i<=3; i++)
   {
      Work w = DoJob(i);
      p.Income += w.CalculateSalary();
   }
   ReportPrinter rp = new ReportPrinter();
   Thread.Sleep(1000); //give it a sec
   //suppose GC occurs at this point
   rp.PrintSalaryReport(p);
}
```



© 2008 Phi Consulting

Phi Consulting

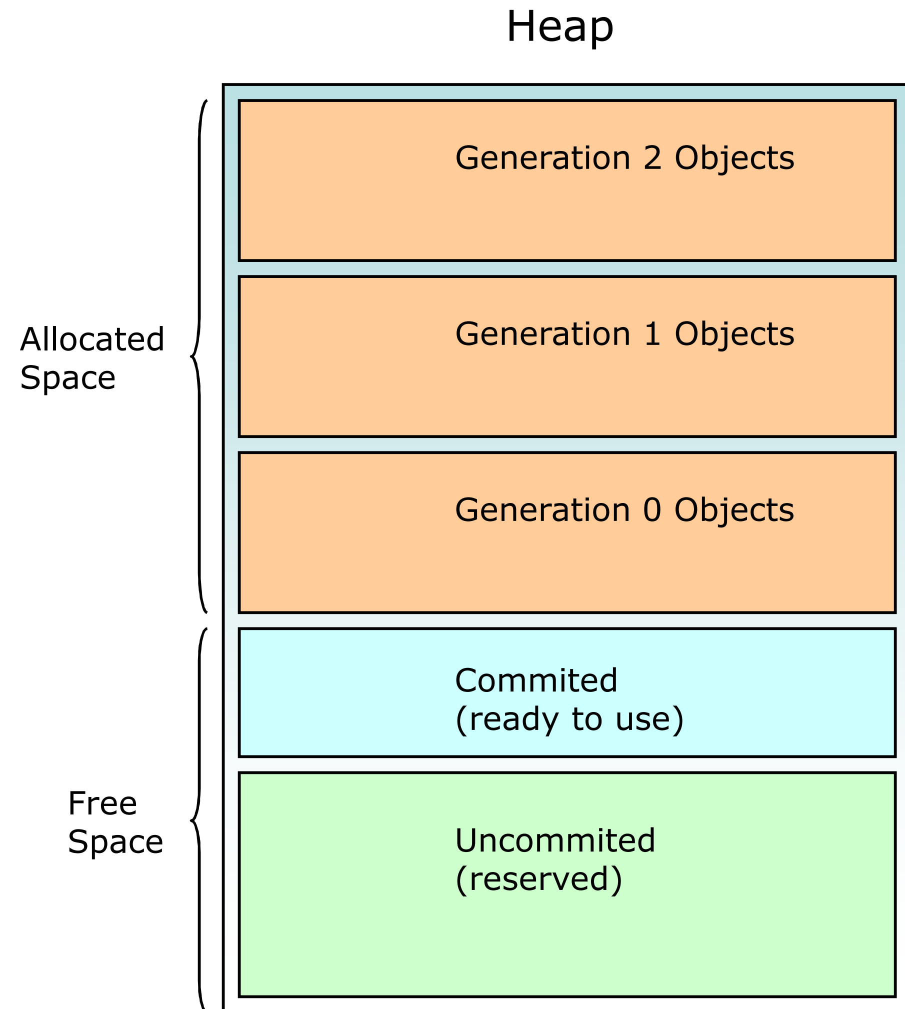
Generations

- It is costly to walk through all the objects in the heap all the time
- So the GC is actually generational – there are 3 generations – gen0, gen1 and gen2
- Objects that survive gen0 collection are promoted to gen1
- Objects that survive gen1 collection are promoted to gen2
- gen0 objects are collected more frequently
- gen1 objects are collected less frequently than gen0 (10-100x less)
- gen2 objects are collected less frequently than gen1 (10-100x less)

© 2008 Phi Consulting

GC Simplified Model

- gen0 are newly allocated objects that have never been considered for collection (typically local variables)
- gen1 have survived a single GC (like forms or lists)
- gen2 have survived multiple collections (like applications)
- When the runtime needs memory it performs gen0 first
- If it needs more memory then it collects gen1
- If that doesn't free up enough memory it will then collect gen2
- The oldest objects are at the lowest addresses, while new objects are created at increasing addresses
- There are never any gaps between objects in the heap
- Only some of the free space is *committed.* When necessary, more memory is acquired from the operating system in the *reserved* address range



© 2008 Phi Consulting

Phi Consulting

Finalization

- The GC supports a concept known as Finalization (a.k.a. Destructors)
- Finalizers allow resource cleanup before the object is collected
- When obj is allocated the runtime adds the ref to a list of objects that require finalization
- When GC occurs if the obj has no references but is contained on the finalization list it is marked ready for finalization
- After GC completes, the finalizer thread wakes and calls the finalizer for all objects in the list
- After the finalizer is called, the obj is removed from the list which will make the object collectible on the NEXT GC collection

© 2008 Phi Consulting Proprietary and Confidential

Phi Consulting

Finalization

- Conclusions:
  - Objects with finalizers have more overhead in the system and hang around longer
  - Finalization takes place on a separate thread from execution
  - There is no guaranteed order of finalization
    - If object a has a ref to object b and both have finalizers, b’s finalizer might run before object a’s, therefore object a might or might not have a valid object b during finalization
  - Finalizers aren’t called on normal program exit, to speed up exit
- Due to all these limitations Finalizers are DISCOURAGED
- You can control the GC to a point
  - To force a collection: GC.Collect();
  - To force finalization on exit: GC.RequestFinalizeOnShutdown();
    - (this may slow down shutdown of the app)
  - To suppress finalization: GC.SuppressFinalize();
    - (removes object from finalization list)

© 2008 Phi Consulting

Phi Consulting

What’s wrong with this code?

```
class Test
{
   public Test() { }
   ~Test() { } //no implementation – empty finalizer
}
```

- Having an empty finalizer forces this potentially gen0 object to be collected as a gen1 object, just by having a finalizer
- This is a severe performance hit

Phi Consulting

Recommended Pattern

- A much preferred cleanup option is implementing the IDisposable interface:

```
class X:  IDisposable
{
  public X(…)   {   … initialize resources …   }

  ~X() {   … release resources …   }

  public void Dispose()
  {
        // this is the same as calling ~X()
      Finalize();
     // no need to finalize later
      System.GC.SuppressFinalize(this);
  }
}
```

```
class X:  IDisposable
{
  public X(…)   {   … initialize resources …   }

  public void Dispose()
  {
    … release resources …
  }
}
```

- The calling class of X is made responsible for calling Dispose()
- The class is then cleaned up BEFORE it is GC'ed
- The GC will bypass Finalization and just collect the object

© 2008 Phi Consulting

Phi Consulting

Conclusion

- **The .NET garbage collector provides a high-speed allocation service with good use of memory and no long-term fragmentation problems**
- **However it is possible to do things that will give you much less than optimal performance**
- **To get the best out of the allocator you should consider practices such as the following:**
  - Allocate all of the memory (or as much as possible) to be used with a given data structure at the same time
  - Remove temporary allocations that can be avoided with little penalty in complexity
  - Make limited use of finalizers, and then only on "leaf" objects, as much as possible. Break objects if necessary to help with this
- **Use Profilers to conduct memory usage profiles to monitor GC usage and obtain the best performance**

© 2008 Phi Consulting Proprietary and Confidential

Φ Phi Consulting

Agenda

- Helpful Namespaces and Objects
  - System.IO
  - System.Text
  - System.Environment
  - System.Security
  - System.Net
  - System.Configuration
  - System.Math
  - System.Convert

© 2008 Phi Consulting
Proprietary and Confidential

Phi Consulting

System.IO

- Input/Output
- The System.IO namespace contains types that allow reading and writing to files and data streams, and types that provide basic file and directory support
  - Stream
    - FileStream
    - NetworkStream
    - MemoryStream
    - Etc.

© 2008 Phi Consulting

Phi Consulting

System.IO

- It also provides classes that can read and write from Streams
  - TextReader/TextWriter
    - StreamReader/StreamWriter
    - StringReader/StringWriter
    - XmlTextReader/XmlTextWriter
    - Etc.

Phi Consulting

System.Text

- Besides being home to the RegularExpressions namespace, and the StringBuilder class System.Text contains
  - Encoders
  - Decoders
- For converting streams from one encoding type to another
  - ASCII
  - UTF-8
  - Unicode
  - Etc.

© 2008 Phi Consulting

Phi Consulting

System.Environment

- Provides information about, and means to manipulate, the current environment and platform
  - CurrentDirectory
  - OSVersion
  - MachineName
  - UserName
  - GetEnvironmentVariable()
  - GetFolderPath() *(windows, my documents, etc.)*
  - GetLogicalDrives()

© 2008 Phi Consulting Proprietary and Confidential

Phi Consulting

System.Security

- Provides the underlying structure of the common language runtime security system, including base classes for permissions
- Allows to set up front certain permission levels that your class requires so the app won't run without them (Declarative Security)
- Declarative Security prevents security exceptions from being thrown by the app and allows it to gracefully shutdown

Phi Consulting

System.Net

- Provides a simple programming interface for many of the protocols used on networks today
  - HTTP Client/Request/Response
  - TCP/UDP sockets
  - System.Net.CredentialCache.DefaultCredentials will get you the credentials of the code that is running – you can pass those to a web service, for example
  - Also home of the NetworkCredential class

System.Configuration

- Provides classes to access/create configuration files
- Extensible – you can create your own custom configuration items

```
<configuration>
    <appSettings>
        <add key="test" value="true"/>
    </appSettings>
</configuration>

string testMode = ConfigurationManager.AppSettings["test"];
```

Phi Consulting

System.Math

- Static class library of mathematical functions
  - Power
  - SQRT
  - Abs
  - Round
  - Trig
  - Min/Max
  - Etc.

System.Convert

- Static class that converts a base data type to another base data type
  - Contains a constant DBNull representing a database column absent of data; that is, database null
  - IsDBNull() – checks if a value is DBNull
  - ChangeType() – change from any type to another
  - ToBoolean(), ToChar(), ToString(), ToDecimal(), etc.
  - ToBase64String(), FromBase64String()