

Reflection and Attributes

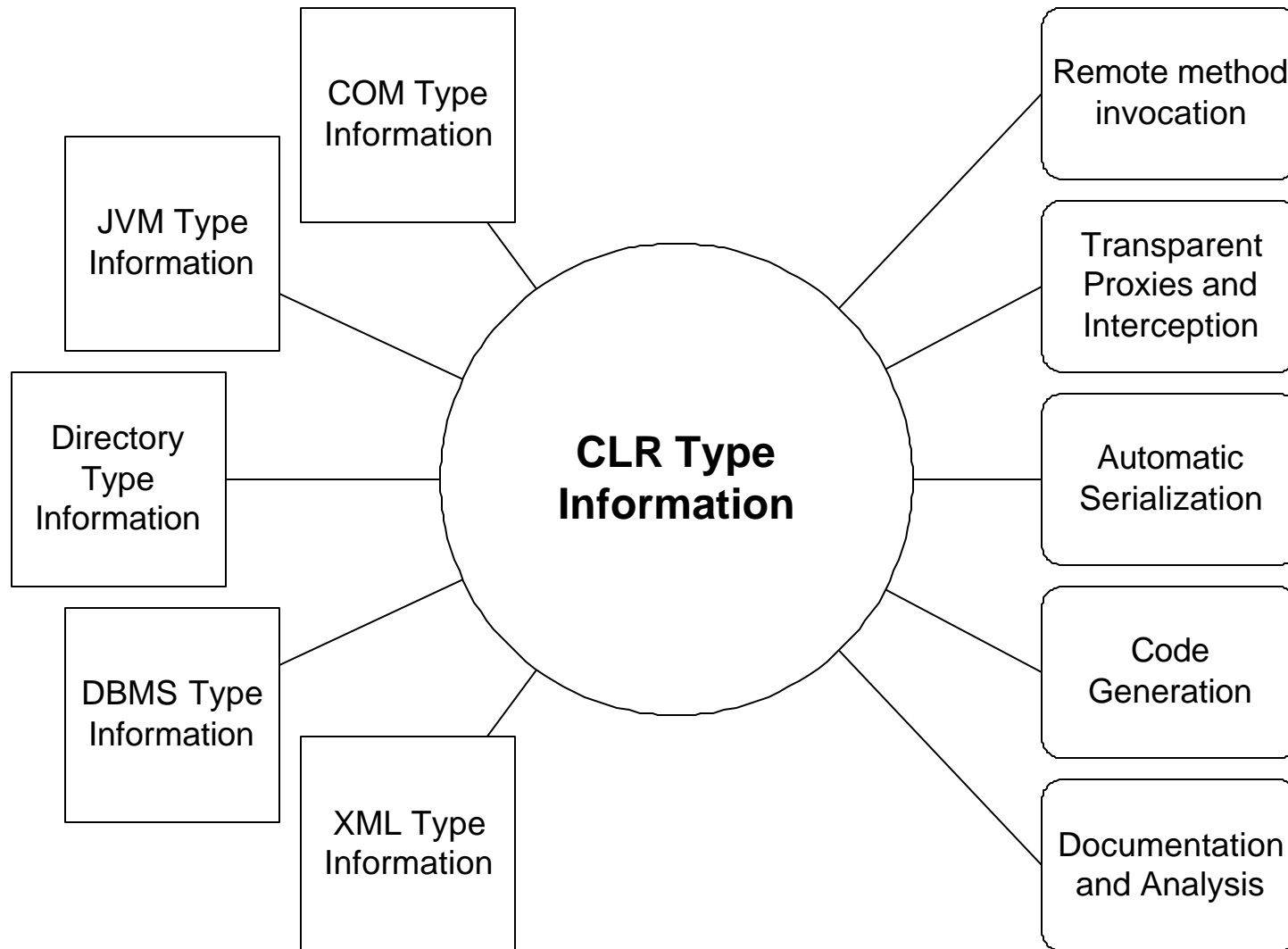


Reflection

- **The Common Language Runtime makes type information ubiquitous, accessible, and extensible**
 - Reflection allows anyone to see any type (barring security)
 - Reflection information extensible via custom attributes
 - Reflection information never optional - all things are knowable
 - **System.Reflection** namespace root of library support
 - Enables numerous runtime-provided services (remoting, serialization, etc.)
 - Enables numerous kinds of tool development (documentation, code generation, etc.)



The role of reflection



Example: using reflection

```
public static void GeneratesSQL(Object obj) {
    Type type = obj.GetType();
    Console.WriteLine("create table {0} (" , type.Name);
    bool needsComma = false;
    foreach (FieldInfo field in type.GetFields()) {
        if (needsComma) Console.WriteLine(", ");
        else needsComma = true;
        Console.WriteLine("{0} {1}", field.Name, field.FieldType);
    }
    Console.WriteLine(")");
}
```

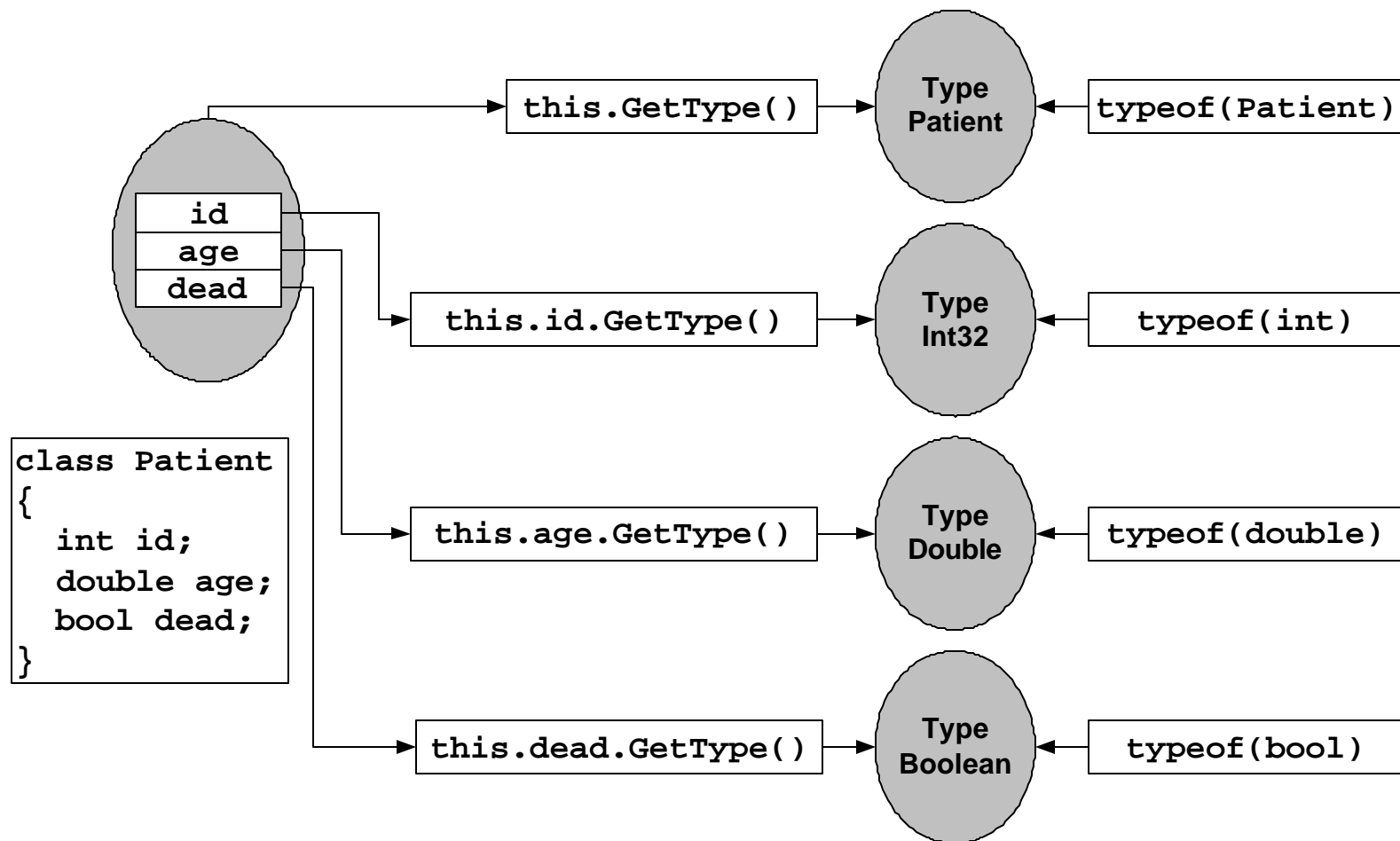


System.Type

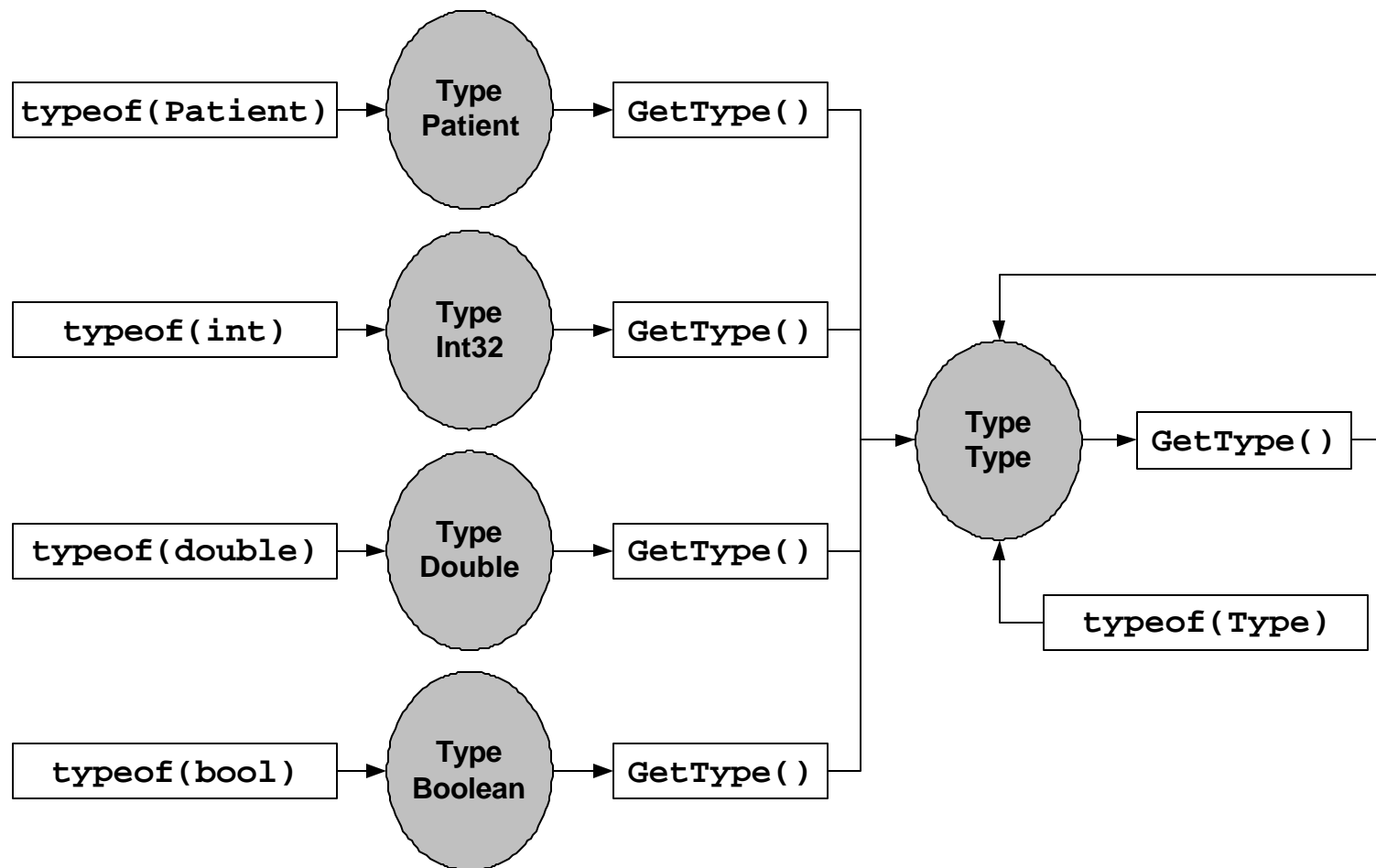
- **System.Type is the focal point of reflection**
 - All objects and values are instances of types
 - Can discover type of object or value using `GetType` method
 - Can reference type by symbolic name using C# `typeof` keyword
 - Types are themselves instances of type `System.Type`
 - Inheritance/compatibility relationships traversable through `System.Type`



Pervasive type and Type.GetType



Pervasive type and System.Type

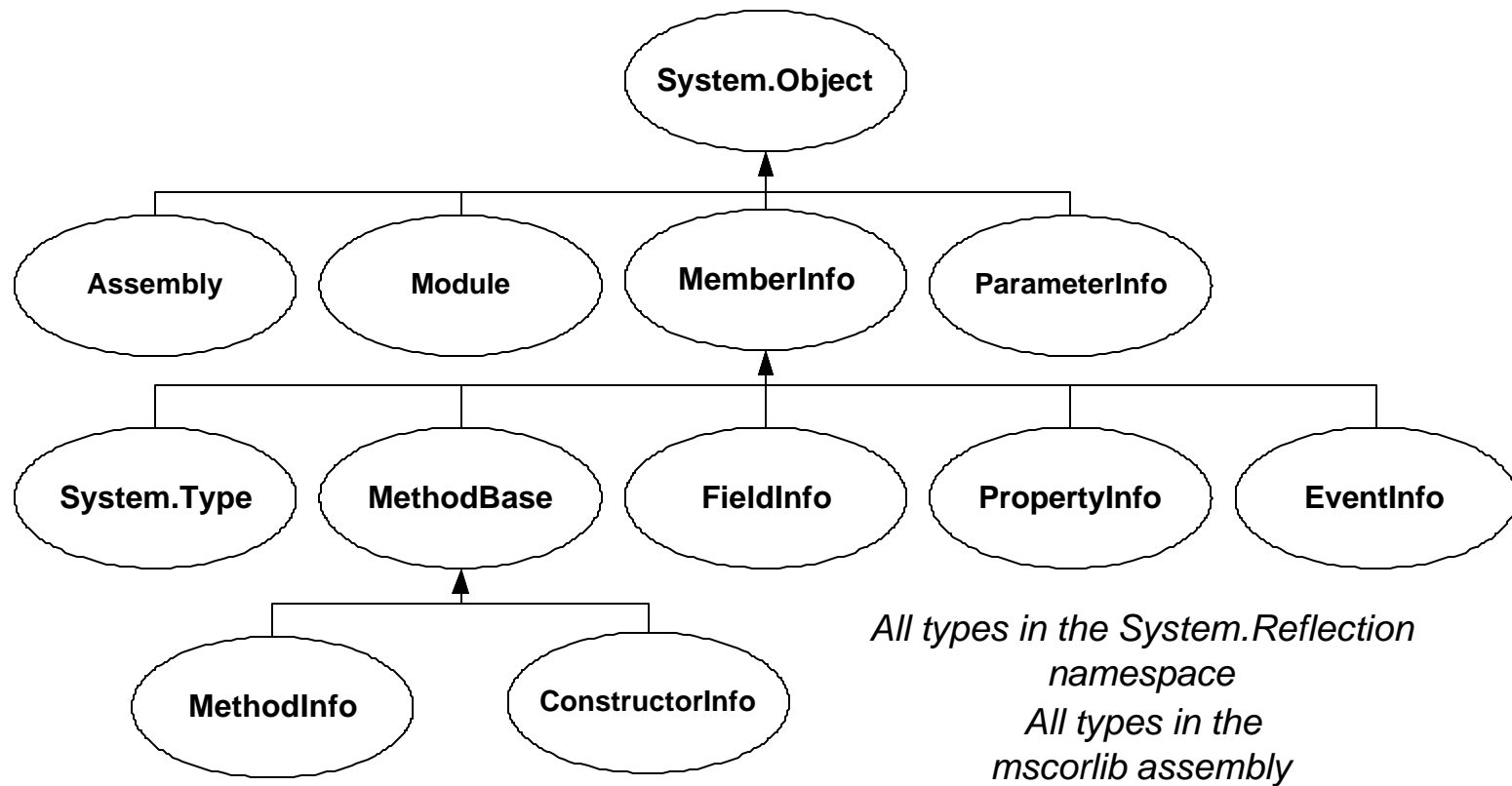


Using System.Type

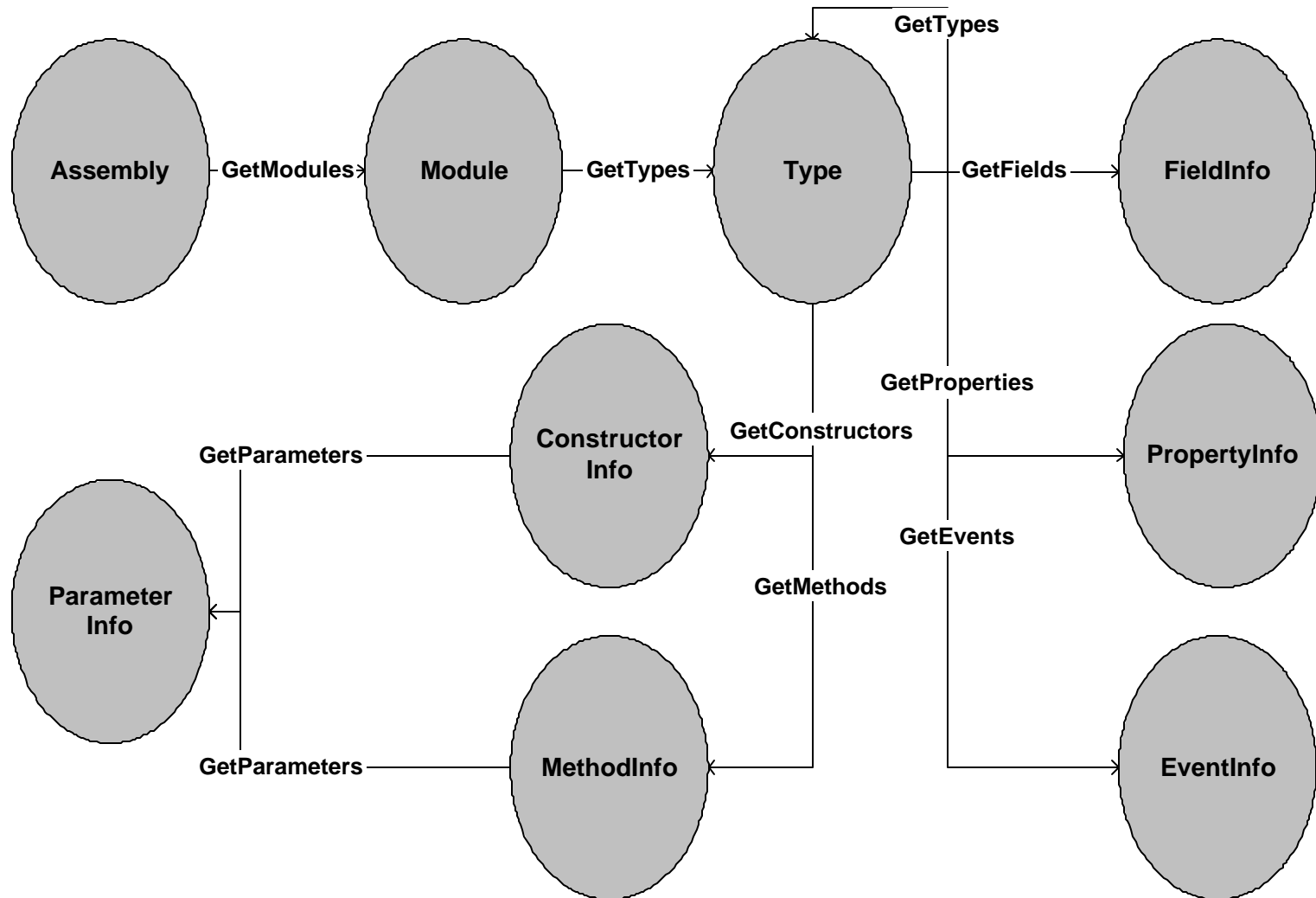
- **All facets of a type are available at runtime**
 - Each kind of member has a reflection class that represents it
 - Most types derive from `System.Runtime.MemberInfo`
 - Can look up members by kind or globally
 - Support for overloading and case-insensitive names for script/VB
 - Can see non-accessible members if security allows



Reflection and the CLR type system



The reflection object model



Example: walking every element in an assembly

```
using System.Reflection;

public static void ListAllMembers(String assemblyName) {
    Assembly assembly = Assembly.Load(assemblyName);
    foreach (Module module in assembly.GetModules())
        foreach (Type type in module.GetTypes())
            foreach (MemberInfo member in type.GetMembers())
                Console.WriteLine("{0}.{1}", type, member.Name);
}
```



Late binding

- **Types may be instantiated and/or members accessed in a late bound manner**
 - Can instantiate type in memory, choosing constructor to call
 - Can read/write fields of an object
 - Can invoke methods
 - Can invoke property getters and setters
 - Public members always accessible
 - Non-public members accessible if callers hold sufficient CAS permissions



Activator.CreateInstance

- `Activator.CreateInstance` **is the late-bound equivalent to operator new**
 - Allocates storage for new type instance
 - Calls specified constructor
 - Returns generic `object` reference
 - Combined with interface-based member access provides flexibility and performance



Example: using Activator.CreateInstance

```
interface ICowboy { void Draw(); }
class Tex : ICowboy { ... }
class Woody : ICowboy { ... }

class App {
    static void Main() {
        Console.WriteLine("Enter western type to use: ");
        string typeName = Console.ReadLine();

        // Late-bound activation
        ICowboy cb = (ICowboy)
            Activator.CreateInstance(Type.GetType(typeName));

        // Early-bound member access:
        cb.Draw();
    }
}
```



Constructor arguments and CreateInstance

```
interface ICowboy { void Draw(); }
class Tex : ICowboy
{
    public Tex( string name, int NumGuns )
    {
        ...
    }
}

class App {
    static void Main() {
        object args[] = { "Jesse James", 4 };
        ICowboy cb = (ICowboy)
            Activator.CreateInstance(typeof(Tex), args);

        cb.Draw();
    }
}
```

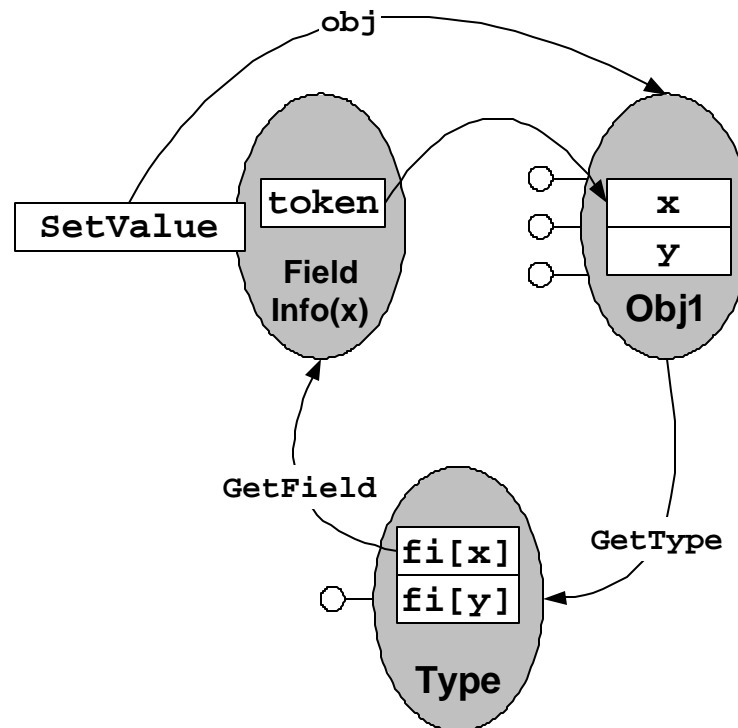


FieldInfo

- **FieldInfo can be used to set/get fields**
 - Field description returned by `Type.GetField`
 - Can see name, type, accessibility and all other aspects of field declaration
 - Can call `GetValue/SetValue` to access the field of an object or type
 - Cannot see field initializers from C# (they are part of IL in constructor)



System.Reflection.FieldInfo



Example: using FieldInfo

```
public static void SetX(Object target, int value) {  
    // Write to field: int x  
    Type type = target.GetType();  
    FieldInfo field = type.GetField("x");  
    field.SetValue(target, value);  
}
```

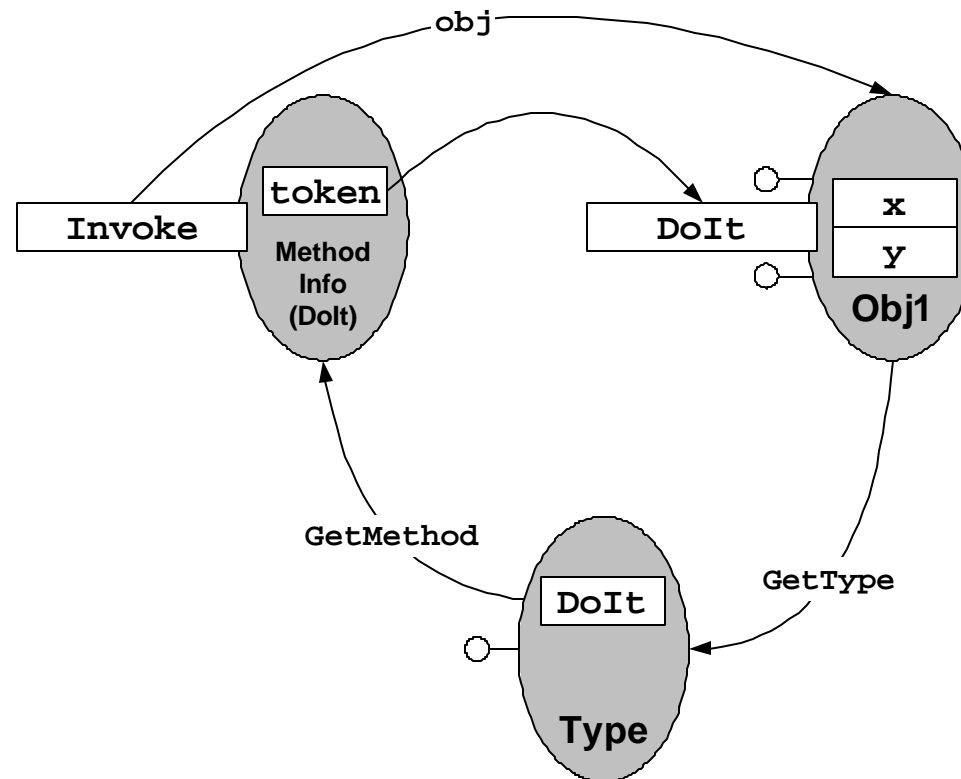


MethodInfo/ConstructorInfo

- **MethodInfo/ConstructorInfo can be used to invoke methods/constructors**
 - Both types derive from common **MethodBase**
 - Method description returned by **Type.GetMethod**
 - Can see name, accessibility, return type, parameters and all other aspects of methods
 - Can call **Invoke** to access the method of an object or type
 - Also accessible from **System.Exception** as well as remoting plumbing



System.Reflection.MethodInfo



Example: using MethodInfo

```
double CallAdd(Object target, double x, double y)
{
    // Call method: double Add( double, double )
    Type type = target.GetType();
    MethodInfo method = type.GetMethod("Add");

    if( method != null ) {
        object[] args = { x, y };
        object result = method.Invoke(target, args);
        return (double)result;
    }
    return(0);
}
```



PropertyInfo

- **PropertyInfo can be used to invoke property getters and setters**
 - Property description returned by `Type.GetProperty`
 - Can see name, type, accessibility and all other aspects of property declaration
 - Can call `GetValue/SetValue` to access the field of an object or type



Example: using PropertyInfo

```
void SetName(Object target, string newName)
{
    // Set property: string Name { set {...} }
    Type type = target.GetType();
    PropertyInfo prop = type.GetProperty("Name");

    if( prop != null ) {
        prop.SetValue(target, newName, null);
    }
}
```



Extending type information

- **CLR type information is extensible using custom attributes**
 - Allows user-defined aspects of a type to be visible via reflection
 - Custom attributes are classes derived from **System.Attribute**
 - C# uses IDL-like syntax with [] prior to the definition of the target
 - Can be comma-delimited or in independent [] blocks
 - Attribute parameters passed by position or name
 - Attributes can be applied to an assembly or module using special syntax
 - Attributes discovered using **IsDefined** and **GetCustomAttributes**



Example: specifying an attribute

```
[assembly: Author("Larry")] // Applies to assembly

[ Author("Moe", "Hi") ]      // Applies to class
class MyClass
{
    // Attribute applies to method:
    [ Author("Curly", Contact="curly@stooges.com") ]
    void f() {
        Object obj = null;
        obj.ToString();
    }
}
```



Example: defining a custom attribute

```
using System;

public class AuthorAttribute : Attribute {
    public string Name;
    public string Comment;
    public string Contact = "";

    public AuthorAttribute( string n, string c ) {
        Name = n;
        Comment = c;
    }

    public AuthorAttribute( string n )
        : this(n, "")
    {
    }
}
```



Example: retrieving custom attributes

```
using System;
using System.Reflection;

void Doctype( string asmName, string typeName )
{
    Type attrType = typeof(AuthorAttribute);

    Assembly a = Assembly.Load(asmName);
    if( a.IsDefined(attrType) ) {
        Console.WriteLine("Assembly has an author");
    }

    Type t = Type.GetType(string.Format("{0}, {1}", typeName, asmName));
    object[] attrs = t.GetCustomAttributes(attrType, false);

    foreach( AuthorAttribute author in attrs ) {
        Console.WriteLine("Name:           " + author.Name);
        Console.WriteLine("Notes:          " + author.Comment);
        Console.WriteLine("Contact info:  " + author.Contact);
    }
}
```

Summary

- **Type information is easily accessible and ubiquitous**
- **Type information allows you to do things at runtime**
- **Type information allows you to do things at development-time**
- **Type information is extensible via custom attributes**

