# Appunti per l'esame Microsoft 70-536

Riccardo Pietrucci

28 settembre 2007

## Indice



# 1 Fondamentali

## 1.1 Il System Type

Il System Type di C# offre due categorie di tipi di dati incorporati (built-in):

**Value Type** contiene direttamente i propri valori. Viene memorizzata nello stack e passata in un metodo tramite una copia. Quando si copia un valore viene creata una seconda copia.

Si utilizza solo se lo spazio necessario alla memorizzazione del tipo non è superiore allo spazio richiesto di un Reference Type[1].

Nei tipi booleani è previsto il solo valore *true* o *false*. E' possibile analizzare anche il caso di valore sconosciuto tramite la forma:

[1] un puntatore occupa 16 byte. Il value type non è conveniente se supera tale valore. Ad esempio un Int32 contiene 4 byte: è possibile creare una struct che contiene fino a 4 interi a 32 bit

```
Nullable<bool> b = null;   // prima forma
bool? b = null;            // seconda forma

bool value;
b.HasValue ? value = b.Value : value = false;
// stesso comportamento con...
value = b.GetValueOrDefault();
value = b.GetValueOrDefault(false);
```

Si può creare un proprio Value Type utilizzando le *struct*. Nel caso in cui si voglia fornire una lista di scelte nell'uso di una classe, si possono creare le *enumerations*.

**Reference Type** contiene un riferimento al valore effettivo. Viene memorizzata nell'heap e passata in un metodo tramite un riferimento allo stesso valore. Quando si copia un riferimento viene creato un secondo puntatore che si riferisce allo stesso valore.

Qualunque tipo che **non** deriva da *System.Value* è un Reference Type. Gli array (*System.Array*) sono Reference.

## 1.2 Interfacce comuni

Interfacce più comuni in .NET:

**IComparable** richiesto per ordinare elementi confrontabili

**IDisposable** rilascia risorse. Utile per oggetti che consumano molte risorse o che devono rilasciare un lock

**IConvertible** consente di convertire a un tipo base tramite i metodi di *System.Convert*.

Le conversioni sono inoltre possibili definendo gli operatori di conversione. Utilizzare la parola chiave *implicit/Widening* per conversioni che non perdono informazioni. Viceversa, utilizzare *explicit/Narrowing* per conversioni che potrebbero perdere informazioni.

**ICloneable** consente di copiare un oggetto

**IEquatable** consente di fare l'override dell'operatore ==

**IFormattable** consente di convertire il valore di un oggetto a una stringa specificatamente formattata

## 1.3 Altre caratteristiche

Una volta inizializzate le stringhe sono *immutabili*: la concatenazione crea le stringhe da concatenare, le concatena e restituisce il risultato distruggendo le altre due. Come dire: anche se la string è un reference type, possiede tutte le caratteristiche che lo fanno "sembrare" un value type. Queste ripetute creazioni e distruzioni comportano penalità di prestazioni, che sono però compensabili tramite StringBuilder. In realtà *non viene però creata una stringa ogni volta*: il CLR crea una Hashtable nella quale memorizza le stringhe già create:

```
string s1 = "Strass"; // ID puntatore: 1
string s2 = "Strass"; // ID puntatore: 1 (Grazie alla CLR)

// invoco s1.Equals(s2): confronto contenuti
s1 == s2; // TRUE

// invoco string.ReferenceEquals(s1, s2): confronto puntatori
(object)s1 == (object)s2; // TRUE
```

Ma che cosa succede se si hanno stringhe dinamiche? Il suddetto metodo non funzionerà più in quanto le stringhe dinamiche non sono inserite nella Hashtable (ovviamente il JIT non le conosce) mentre *Equals* sì. E' quindi bello ma inutile? In realtà è possibile forzare una stringa dinamica ad essere inserita nell'Hashtable attraverso il metodo String.Intern:

```
string s1 = "Strass"; // ID puntatore: 1
string s2 = "Stra";   // ID puntatore: 2

s2 += "ss";  // stringa dinamica

// invoco s1.Equals(s2): confronto contenuti
s1 == s2; // TRUE

// invoco string.ReferenceEquals(s1, s2): confronto puntatori
(object)s1 == (object)s2; // FALSE

s2 = string.Intern(s2);  // ID puntatore: 1
(object)s1 == (object)s2; // TRUE
```

L'attributo *TypeForwardedTo* consente di spostare un tipo da un assembly a un altro.

I *Generics* consentono all'utilizzatore di una classe di specificare il tipo di un elemento per evitare cast espliciti. Nei Generics è possibile imporre sul tipo specificabile dal client vincoli di interfaccia, di classe padre, di costruttore no-args o di tipo (reference o value).

Il *boxing* consente di convertire un Value Type in un Reference Type. L'*unboxing* consente il contrario. Poiché il boxing e l'unboxing consumano risorse, nei Value Type è necessario se possibile utilizzare gli generics e fare l'override dei metodi di Object, in modo tale da impedire il boxing al richiamo dei metodi di Object.

Gli *attributi* descrivono un tipo, un metodo o una proprietà e sono analizzabili tramite le funzioni di *Reflection*. Consentono di specificare la sicurezza, i permessi, le capacità e le meta-informazioni.

Per sollevare un *evento*, è necessario creare un *delegate*[2] (classe che contiene un riferimento a un metodo con una signature specifica), creare un *campo evento* e invocare il delegate utilizzando se necessario un *EventArgs* personalizzato:

```
class SampleControl: Component {
   public event MouseEventHandler MouseDown;

   protected virtual void OnMouseDown() {
      // Make a temporary copy of the event to avoid possibility of
      // a race condition if the last subscriber unsubscribes
      // immediately after the null check and before the event is raised
      EventHandler<CustomEventArgs> handler = RaiseCustomEvent;

      // Event will be null if there are no subscribers
      if (handler != null) {
         handler(this, null);
      }
   }
}
```

Il delegate può anche restituire un risultato, ma verrà ottenuto solo il risultato restituito dall'ultimo oggetto notificato. Non è quindi utile restituire un risultato in comunicazioni multicast.

Per impostazione predefinita, quando una classe dichiara un evento, *il compilatore assegna della memoria a un campo per l'archiviazione delle informazioni sull'evento*. Se una classe comprende molti eventi inutilizzati, essi consumeranno inutilmente memoria. Per questi casi, in .NET Framework è disponibile un costrutto denominato *proprietà evento* che è possibile utilizzare con un'altra struttura dati a scelta per archiviare i delegati degli eventi.

[2] utile il metodo statico *Delegate.Combine*, che restituisce un delegate come concatenazione dei delegate passati per parametro nell'ordine specificato.

Le proprietà evento sono composte da dichiarazioni di eventi accompagnate dalle funzioni di accesso agli eventi. I metodi di accesso agli eventi sono metodi che lo sviluppatore definisce per consentire l'aggiunta e la rimozione di istanze dei delegati di evento dalla struttura dati di memorizzazione. Si noti che le proprietà evento sono più lente dei campi evento in quanto prima di poter chiamare un delegato, è necessario recuperarlo. Si pone pertanto la necessità di scegliere tra risparmio di memoria e velocità[3]. Se la propria classe definisce molti eventi che vengono generati di rado, è possibile che si preferisca implementare le proprietà evento. I controlli Windows Form e i controlli server ASP.NET utilizzano le proprietà evento anziché i campi evento. Uno degli approcci più comuni consiste nell'implementare un insieme di delegati indicizzato da una chiave evento tramite una *EventHandlerList*:

```
class SampleControl: Component {
   protected EventHandlerList events = new EventHandlerList();

   // Define a unique key for each event.
   static readonly object mouseDownEventKey = new object();
   static readonly object mouseUpEventKey = new object();
.
   public event MouseEventHandler MouseDown {
      add { events.AddHandler(mouseDownEventKey, value); }
      remove { events.RemoveHandler(mouseDownEventKey, value); }
   }

   public event MouseEventHandler MouseUp {
      add { events.AddHandler(mouseUpEventKey, value); }
      remove { events.RemoveHandler(mouseUpEventKey, value); }
   }

   protected virtual void OnMouseDown() {
      MouseEventHandler handler = events[mouseDownEventKey];
      if(handler != null) {
         handler(this, null);
      }
   }
}
```

# 2 Input/Output (I/O)

## 2.1 FileSystem

Le classi del FileSystem sono separate in due tipi:

**Informazioni** la maggior parte delle informazioni derivano dalla classe base FileSystemInfo, tra cui troviamo FileInfo, DirectoryInfo.

**Utilità** metodi statici che consentono di eseguire operazioni su oggetti del filesystem, tra cui troviamo File, Directory, Path. Ad esempio troviamo *Path.GetTempFileName* che restituisce il percorso completo di un file temporaneo creato appositamente dal sistema. Oppure troviamo *File.Move* che consente di spostare (o rinominare) un file e solleva IOException nel caso in cui il file già esista. *FileInfo.Length* restituisce la dimensione del file in byte. O ancora troviamo il metodo statico Directory.CreateDirectory che esegue un check di Directory.Exists seguito da una creazione (praticamente un touch).

*DriveInfo* non deriva da FileSystemInfo e contiene sia metodi informazionali che metodi statici di utilità, come il metodo statico *DriveInfo.GetDrives()* che restituisce vari drive il cui tipo è verificabile analizzando la proprietà di istanza DriveInfo.DriveType, che può essere CDRom, Fixed (disco rigido), Removable, Network, Ram.

[3]anche se in genere è comunque preferibile utilizzare proprietà evento, a meno che non abbiamo pochissimi eventi...

*FileSystemWatcher* consente di monitorare cambiamenti nelle directory del filesystem tramite gli eventi Changed, Created, Deleted, Renamed, Error. Per abilitare gli eventi, è necessario impostare a true la proprietà *EnableRaisingEvents* (stesso discorso anche per EventLog). Non è invece necessario abilitare questa proprietà in caso di richieste sincrone, disponibili in FileSystemWatcher.*WaitForChanged*, che richiede un WatcherChangeTypes.

## 2.2 Stream

Tutti i flussi di dati contengono le stesse informazioni di base e lo stesso modo di lavorare con i dati. Tali informazioni sono ereditate dalla classe base *Stream* per tutti i tipi di flussi. Le classi derivate sono le seguenti:

**FileStream** funzioni per aprire, leggere e scrivere flussi di file. Per ottenere un FileStream si possono utilizzare le funzioni di utilità di File.

**MemoryStream** consente di memorizzare temporaneamente per riversarlo ad esempio in un file tramite *WriteTo* o in un'array di byte tramite *ToArray*. Utile per limitare il lock di un file al momento della scrittura. Non richiedono alcun buffer.

Il problema in questo caso è che verrebbe inserito tutto il file prima di riversarlo in un file. Per trasferire incrementalmente i dati di un buffer limitato, si utilizza *BufferedStream*, che avvolge uno Stream per effettuare la scrittura effettiva sullo Stream avvolto solo quando il buffer è pieno[4].

**CryptoStream**

**NetworkStream**

**GZipStream** avvolge il flusso di destinazione per comprimere i dati da scrivere fino a 4 GB dati utilizzando la compressione GZIP. In fase di decompressione consente un data corruption check.

**DeflateStream** avvolge il flusso di destinazione per comprimere i dati da scrivere fino a 4 GB dati utilizzando la compressione DEFLATE. Rispetto al GZIP è privo di header, non utilizza ridondanza ciclica ed è conveniente solo se utilizzato nell'ambito del framework .NET.

Gli stream di compressione (GZip e Deflate) avvolgono un flusso. Nel costruttore possiedono alla fine un valore booleano finale che indica se alla chiusura dello stream di compressione deve essere cil flusso comunque lasciato aperto il flusso avvolto (di default: false).

Gli Stream leggono i flussi come byte. Per facilitare la lettura si utilizzano le classi astratte *TextReader* e *TextWriter* che avvolgono uno Stream e leggono i flussi come un tipo specifico, non come generici byte. Implementazioni di queste classi sono *StreamReader*, *StringReader*, *BinaryReader*.

## 2.3 Isolated Storage

L'Isolated Storage è uno speciale path di una specifica utenza in cui l'utente ha sempre diritto di scrittura. Per creare file e cartelle nell'Isolated Storage si utilizza la classe

[4]FileStream è già un BufferedStream

*IsolatedStorageFile* che all'apertura dei file restituisce un *IsolatedStorageFileStream*, che deriva da FileStream. Per creare un IsolatedStorageFile esistono differenti metodi statici: *Get(User|Machine)StoreFor(Assembly|Application|Domain)*. I due tipi di visibilità più usati:

**Assembly/Machine** informazioni sull'assembly in esecuzione. Normalmente è localizzato in:

```
C:\Programmi\$ASSEMBLY
```

**Assembly/User** informazioni sull'utente corrente dell'assembly in esecuzione.

In sistemi Windows XP aggiornati da ambienti Windows 2000 che utilizzano profili roaming è localizzato in:

```
$SYSTEMROOT:\Document And Settings\$USER\
              \Application Data\$ASSEMBLY
```

In sistemi Windows XP non aggiornati senza profili roaming è localizzato in:

```
$SYSTEMROOT:\Document And Settings\$USER\
    Local Settings\Application Data\$ASSEMBLY
```

In sistemi Windows 2000, Windows XP e Windows Server 2003 aggiornati da ambienti Windows NT che utilizzano profili roaming è localizzato in:

```
$SYSTEMROOT:\Profiles\$USER\
              \Application Data\$ASSEMBLY
```

In sistemi Windows 98 e Windows ME senza profili utente è localizzato in:

```
$SYSTEMROOT:\Application Data\$ASSEMBLY
```

Affinché un assembly possa far uso dell'isolated storage, è necessario garantire i permessi (demand) annotando la classe o il metodo tramite *IsolatedStorageFilePermission*. Per visualizzare o rimuovere gli isolated storage utilizzare *storeadm.exe*.

# 3 Testo

## 3.1 Espressioni Regolari

Le espressioni regolari consentono di determinare se un testo corrisponde a un formato. I caratteri speciali più comuni sono:

**^** inizio di una stringa

**$** fine di una stringa

**.** qualunque carattere

**?** carattere precedente opzionale

***** carattere precedente può avere da 0 a $n$ occorrenze

Per verificare un testo bisogna creare un pattern (*Regex.Match*) utilizzando gruppi (proprietà *Match.Groups*, identificate da '(' e ')') per estrapolare le porzioni di testo interessate. Per un maggior controllo, si può passare come parametro un *RegexOptions*. Per verificare semplicemente la conformità dell'input senza restituirne i risultati come istanza Match è possibile utilizzare il metodo *Regex.IsMatch*.

## 3.2 Codifica

Le seguenti codifiche esistono:

**ASCII** 7 bit. Solo alfabeto latino.

**ANSI (ISO 8859)** 8 bit. I primi 128 caratteri sono dello standard ASCII, mentre i rimanenti sono code page specifici del linguaggio in uso. Ad esempio "ISO-8859-1" (Western European) corrisponde al code page 28591

**Unicode** non specifica un encoding type. Esistono UTF-32 e UTF-16. UTF-8 usa 8 bit, 16 bit, 24 bit fino a 48 bit. Il framework .NET utilizza la codifica UTF-16. Esiste anche UTF-7, ma per utilizzare questa codifica serve dichiararlo esplicitamente. UTF-8 contiene anche caratteri cinesi, giapponesi e coreani.

*Encoding.GetEncoding* (statico) restituisce una particolare codifica. Il metodo *Encoding.GetBytes* converte una stringa unicode nella rappresentazione a byte secondo la codifica specificata. È possibile specificare l'Encoding nel costruttore di uno Stream.

# 4 Collezioni

## 4.1 Collezioni standard

In .NET ogni collezione è una *ICollection*, che a sua volta deriva da *IEnumerable* per la scansione di una lista. L'interfaccia IEnumerable abilita l'iterazione *foreach* e stabilisce che deve esistere un metodo *GetEnumerator* che restituisce un'istanza *IEnumerator* per la scansione di una lista. La collezione ICollection possiede la proprietà readonly *SyncRoot* ereditata da tutte le collezioni, che restituisce un oggetto che può essere utilizzato per sincronizzare l'accesso a ICollection per impedire che altri thread possano modificare contemporaneamente l'istanza dell'insieme[5]:

```
ICollection c = ...
lock(c.SyncRoot) {
  // Some operation on the collection,
  //         which is now thread safe.
}
```

Il namespace System.Collections supporta diversi tipi di collezioni:

**IList** collezione di elementi. Supporta metodi come Add, AddRange, Insert, Remove, RemoveRange, RemoveAt, Clear, IndexOf, Contains

- **ArrayList** collezione dinamicamente ridimensionabile basata su indici. Supporta il metodo Sort, che utilizza un IComparer per il confronto.

**Queue** collezione FIFO. Supporta metodi Enqueue, Dequeue, Peek, Count.

[5] osservando il codice, si potrebbe pensare di scrivere semplicemente "`lock(c){...}`", ma questo *è sconsigliato in implementazioni di interface*. In particolare, ICollection spesso avvolge altre implementazioni ICollection (ad esempio un HybridDictionary avvolge una Hashtable) quindi eseguire il lock sull'intera istanza non abilita il lock sull'istanza avvolta. Attraverso questo espediente, invece, è possibile avvolgere anche la proprietà SyncRoot (HybridDictionary.SyncRoot in realtà restituisce un Hashtable.SyncRoot) consentendo di eseguire il lock sulla implementazione di ICollection concretamente in uso (Hashtable), e non sulle sue decorazioni (HybridDictionary)

**Stack** collezione LIFO (pila). Supporta metodi Pop, Push, Peek, Count.

**IDictionary** generica mappa. Supporta metodi Add. Memorizza gli elementi come oggetti *DictionaryEntry*. Consente di ottenere valori tramite indicizzatori passando la chiave. Supporta il metodo TryGetValue(key, out value) che esegue ContainsKey e in caso affermativo ottiene il valore memorizzandolo nel parametro value: utile per minimizzare l'accesso alla mappa.

**SortedList** mappa ordinata secondo l'ordinamento naturale o, in alternativa, secondo l'ordinamento specificato da un IComparer.

**Hashtable** mappa ordinata tramite metodi GetHashCode e Equals o, in alternativa, tramite l'interfaccia *IEqualityComparer*. Supporta metodi ContainsKey e ContainsValue e consente la scansione ordinata per hash

**ListDictionary** efficiente collezione indicata per memorizzazione di pochi elementi, da utilizzare in alternativa all'Hashtable

**HybridDictionary** usa un ListDictionary quando ha pochi elementi. Se necessario, migra la collezione a una Hashtable per memorizzare numerosi elementi

**OrderedDictionary** mappa con ordinamento di inserzione

La classe di utilità *CollectionsUtil* fornisce scorciatoie di utilità alle collezioni, come la creazione di collezioni con un'ordinamento case-insensitive.

### 4.2 Collezioni specializzate

Utilizzare le collezioni elencate forza a un cast esplicito per ottenere il tipo desiderato. Esistono collezioni specializzate che lavorano con uno specifico tipo e consentono operazioni specializzate su di esso:

**BitArray** lista ridimensionabile di Boolean. Non deriva da IList: i metodi Add e Remove non sono supportati. Consente operazioni booleani su due istanze BitArray delle stesse dimensioni. *Non* è *dinamicamente* ridimensionabile

**BitVector32** intero a 32 bit. Utile per gestire ogni singolo bit dell'Int32, creare bit mask e/o creare sezioni per il bit packing[6]

**StringCollection** ArrayList di string

**StringDictionary** Hashtable di string-string

**NameValueCollection** mappa `string-IList<string>`. Consente la memorizzazione di una *lista* di valori per una determinata chiave, ottenibile tramite *GetValues*.

### 4.3 Collezioni generiche

Per eliminare completamente il cast si utilizzano le collezioni generiche. Le collezioni disponibili e il loro legame con le versioni standard sono mostrate in tab.1. L'elemento di novità è la *LinkedList*, un insieme di elementi *LinkedListNode* collegati a catena, in cui è possibile accedere ai propri vicini senza conoscere la collezione nella sua interezza.

| Tipo | Tipo Generico |
|---:|---|
| ArrayList | List |
| Queue | Queue |
| Stack | Stack |
| Hashtable | Dictionary |
| SortedList | SortedList |
| ListDictionary | Dictionary |
| HybridDictionary | Dictionary |
| OrderedDictionary | Dictionary |
| SortedDictionary | SortedDictionary |
| NameValueCollection | Dictionary |
| DictionaryEntry | NameValuePair |
| StringCollection | List<String> |
| StringDictionary | Dictionary<String> |
| N/A | LinkedList |

Tabella 1: Tipi Generici Equivalenti

A prima vista potrebbe sembrare che SortedList e SortedDictionary siano due classi equivalenti. In realtà esistono alcune differenze:

- SortedList consente di ottenere elementi tramite indici mentre SortedDictionary no. Infatti le proprietà SortedList.Keys e SortedList.Values sono indicizzate
- SortedList ha inserimento con complessità $O(1)$ e rimozione con $O(n)$. In SortedDictionary, invece, sia inserimenti che rimozioni hanno complessità $O(\log(n))$
- SortedList usa meno memoria di SortedDictionary

## 5 Serializzazione

### 5.1 Tecniche di serializzazione a runtime

Il namespace System.Runtime.Serialization consente di convertire lo stato di un oggetto in un formato esportabile (sequenza di byte).

Il supporto nativo binario viene dalla classe *BinaryFormatter* e dai metodi d'istanza *Serialize* e *Deserialize*. Nel caso di collegamenti verso altri oggetti ci potrebbe essere il rischio di (de)serializzazioni multiple, ma internamente l'*IFormatter* interroga l'*ObjectManager* per ottenere se è un backward reference (già fatto) o un forward reference (non ancora fatto: registra un fixup) per risolvere i riferimenti. Per ottenere migliori performance si può utilizzare UnsafeDeserialize, più veloce ma che utilizza codice unmanaged e richiede quindi maggiori permessi.

Per abilitare la serializzazione è necessario aggiungere l'attributo *Serializable* alla classe, che non è ereditabile e deve essere assegnato in modo esplicito alle classi derivate. La serializzazione memorizza tutti i dati, anche quelli privati. Ciò apre possibili attacchi di sicurezza e di default è abilitato solo per esecuzione locale, altrimenti è necessario aggiungere l'attributo *SecurityPermission* con il flag *SerializationFormatter* specificato. Per disabilitare la serializzazione è necessario aggiungere l'attributo *NonSerialized*, utile per valori il cui stato

[6] per memorizzare ad esempio i valori 10, 50 e 500 in un singolo Int32

è calcolabile. Per (ri)memorizzare variabili non serializzate implementare l'interfaccia *IDeserializationCallback* implementandone il metodo *OnDeserialization*. Nel caso di versioni incompatibili (sto deserializzando dalla versione 3.0 mentre l'attuale è la 3.1) si potrebbero verificare eccezioni nel caso in cui un attributo della versione 3.1 non esiste nella 3.0. Per questo è utile utilizzare l'attributo *OptionalField* nei nuovi membri che potrebbero causare problemi di versioning e, se necessario, conferire valori di default tramite IDeserializationCallback.

### 5.2 Serializzazione XML

BinaryFormatter non è l'unico formato possibile. Esistono infatti il *SoapFormatter* e l'*XmlSerializer*[7]. Quest'ultimo richiede che la classe da serializzare sia pubblica, che tutti i membri siano pubblici e che sia dotata di un costruttore privo di argomenti. E' possibile controllare la serializzazione tramite gli attributi: XmlRoot, XmlAttribute, XmlIgnore, XmlArray, XmlArrayItem. L'eseguibile `xsd.exe` consente di analizzare in input un XML Schema (*.xsd) per produrre la rappresentazione a oggetti dotata di tutti quegli attributi che consentono di mantenersi conforme allo schema in caso di serializzazione. Tutti i dataset, gli array, le collezioni e le istanze di XmlElement o XmlNode possono essere serializzate con l'XmlSerializer. Tutti i nodi XML non deserializzabili vengono ignorati a meno che non si utilizza l'evento XmlSerializer.UnknownNode, che si verifica quando si incontra un nodo XML di tipo sconosciuto per gestirlo opportunamente[8]. Se si vuole essere più specifici, si utilizzano gli eventi UnknownElement e UnknownAttribute che discriminano i due casi (node comprende sia element che attribute). Se invece un tipo è riconosciuto, ma non è utilizzato allora si utilizza l'evento UnreferencedObject.

### 5.3 Serializzazioni Personalizzate

Per avere un controllo completo si può implementare l'interfaccia *ISerializable* (o IXmlSerializable nel caso xml). Per la serializzazione è necessario implementare il metodo virtuale *GetObjectData* abilitandone la sicurezza, mentre per la deserializzazione è necessario un costruttore specifico. Entrambi i metodi prendono come parametri istanze *SerializationInfo* e uno *StreamingContext*. Il SerializationInfo converte i valori di tipi nativi utilizzando un'istanza conforme all'interfaccia IFormatterConverter, mentre lo StreamingContext fornisce informazioni sulla destinazione dell'oggetto serializzato (CrossProcess, CrossMachine, File, Persistence, Remoting, ...)

In alternativa, si può controllare il BinaryFormatter utilizzando attributi OnSerializing[9], OnSerialized, OnDeserializing, OnDeserialized che accettano una procedura con un parametro StreamingContext. Inoltre, la proprietà BinaryFormatter.FilterLevel consente di specificare il livello di deserializzazione automatica per i servizi remoti di .NET:

**Full** supporta la deserializzazione automatica di tutti i tipi che i servizi remoti supportano in tutte le situazioni

**Low** attiva la protezione contro gli attacchi di deserializzazione tramite la deserializzazione dei soli tipi associati alla funzionalità di base dei servizi remoti

Per creare rapidamente un formatter personalizzato è necessario implementare IFormatter o IGenericFormatter e utilizzare funzioni di utilità nella classe FormatterServices per facilitarne l'implementazione.

## 6 Grafica

Per creare grafica la classe System.Drawing.*Graphics* fornisce metodi per disegnare per un supporto grafico. La classe *Pen* consente di disegnare linee e curve nei metodi Draw* di Graphics mentre le classi che derivano dalla classe astratta *Brush* consentono di riempire una forma geometrica nei metodi Fill* di Graphics. Per visualizzare le immagini create in una GUI è necessario utilizzare il componente *PictureBox*. Le struct più utilizzate sono *Color*, *Point*, *Rectangle* e *Size*.

Per creare un'istanza di Graphics è possibile utilizzare il metodo CreateGraphics nei componenti GUI o crearla a partire da una classe *Image*, di cui un'implementazione concreta è la classe *Bitmap*[10], che contiene inoltre i due metodi GetPixel e SetPixel. La classe Image contiene metodi statici di utilità per la creazione di immagini e il salvataggio. Nel metodo Image.Save consente anche di specificare l'ImageFormat (di default jpg).

La classe *SystemIcons* consente di ottenere le icone di sistema, utili per dialog di errore o di warning o in altri contesti.

E' possibile inserire testo in una *Graphics* creando un oggetto *Font* specificandone la *FontFamily* e la dimensione e successivamente invocando il metodo di Graphics *DrawString* che accetta un font e opzionalmente un'istanza Brush e uno *StringFormat* per controllarne la formattazione.

Si noti che il sistema di coordinate viewport (quello utilizzato dal namespace Graphics) ha origine in alto a sinistra. L'ascissa si comporta nel modo usuale: valori maggiori considerano valori più a destra, mentre l'ordinata si comporta nel modo contrario di come siamo abituati ad usarla: valori maggiori considerano valori più in basso (essendo l'origine in alto).

## 7 Threading

### 7.1 Creare Thread

Il namespace *System.Threading* contiene i tipi che sono usati per creare e gestire thread multipli in un'applicazione. La classe *Thread* consente di creare e avviare thread. Per creare un Thread è necessario passare un delegate (*Parameterized)ThreadStart* che contiene il metodo da avviare al momento dell'invocazione del metodo *Thread.Start*. Il metodo *Thread.Sleep* consente di sospendere un thread per consentire il processamento degli altri thread mentre il metodo *Thread.Interrupt* forza l'interruzione dello sleep attraverso il lancio dell'eccezione ThreadInterruptedException. mentre il metodo *Thread.Join* consente di attendere la terminazione di un thread. La proprietà *Thread.Priority* consente di specificarne la priorità di schedulazione mentre la proprietà

[7] quest'ultimo richiede di specificare il Type della classe da serializzare

[8] utile in casi d'uso in cui esistono modifiche XML che non necessariamente si ripercuotono sulla struttura ad oggetti, poiché i due cambiamenti non viaggiano parallelamente

[9] che si verifica *prima* della serializzazione

[10] Bitmap è un oggetto utilizzato per operare con immagini definite dai dati pixel, non c'entra niente con l'estensione BMP: si può scrivere infatti `Bitmap b = new Bitmap(foo.jpg)`

*Thread.ThreadState* consente di ottenerne lo stato (Unstarted, Running, ...). La proprietà *Thread.BasePriority* consente di ottenere la proprietà tramite un valore numerico: 4 è bassa priorità, 8 è normale, 13 è alta e 24 è real-time.

Il meccanismo principale per terminare un thread è tramite *Thread.Abort*, che fa sollevare una *ThreadAbortException* verso il thread chiamante. Si può gestire un Abort catturando tale eccezione per verificare e correggere stati inconsistenti. In alternativa, è possibile utilizzare i metodi statici *Thread.BeginCriticalRegion* e *Thread.EndCriticalRegion*: all'interno di una regione critica non è possibile interrompere un thread. Il metodo Abort attenderà di trovarsi in uno stato safe per procedere.

Ogni thread ha informazioni associate, come sicurezza, localizzazione e transizioni che vengono propagate a thread ausiliari e che sono utili a condividere informazioni tra i vari thread. E' possibile interrompere tale propagazione e ripristinarla attraverso i metodi statici di *ExecutionContext*: *SuppressFlow* e *RestoreFlow*. E' possibile catturare il contesto corrente tramite ExecutionContext.*Capture* (S) ed eseguire un delegate *ContextCallback* tramite ExecutionContext.*Run* (S) per eseguire codice arbitrario secondo un contesto specifico.

## 7.2 Lock di sincronizzazione

Una semplice istruzione come `i++;` in realtà nasconde tre operazioni differenti:

1. caricare il valore attuale di $i$ nel registro del processore
2. incrementare il valore nel registro
3. ricopiare il valore dal registro alla memoria

Se più thread dovessero contemporaneamente incrementare tale valore, il risultato è inconsistente: qualche incremento potrebbe perdersi. Per evitare questo comportamento si utilizza l'operazione atomica `Interlocked.Increment(ref i)`. In teoria il modificatore *volatile* sarebbe sufficiente per ovviare al problema: previene il caching di dati nella CPU, ma a seconda del modello di CPU o di memoria potrebbe non bastare. Per ottenere il comportamento corretto a prescindere dal modello del calcolatore è necessario utilizzare le funzioni di Interlocked. Altre funzioni sono utili in Interlocked, tra cui Add, Decrement, Equals, Read, Exchange e CompareExchange, che scambia i due parametri $p_1$ e $p_2$ se $p_1 == p_3$.

Nonostante tali funzioni può risultare necessario non solo svolgere atomicamente singole istruzioni, ma anche blocchi di codice. Per questo si utilizzano i lock di sincronizzazione, utilizzabili tramite la parola chiave *lock*, che richiede un'istanza di un oggetto per l'identificazione del lock. Bisogna passare come oggetto sempre un reference type: non è quindi possibile passare *due distinti* value type boxed[11] o String[12]. Se si necessità di più controllo si può utilizzare la classe *Monitor*, tramite i metodi *Enter* ed *Exit* (sempre nella parte finally). In aggiunta, troviamo il metodo *Wait*, che rilascia il lock e invoca un Thread.Sleep, e il metodo *TryEnter* che prova a creare un lock fino a un timeout specificato, utile per prevenire situazioni di deadlock.

[11] cambiamenti nella variabile non si riflettono nella copia boxed

[12] anche se le String sono reference type, sono immutabili e sovrascrivono il comportamento degli operatori di uguaglianza, comportandosi di fatto come value type

## 7.3 Altri metodi di sincronizzazione

Oltre alle classi Monitor esistono:

**ReaderWriterLock** reader molteplici accedono ai dati contemporaneamente tramite *AcquireReaderLock*, ma solo il writer può acquisire il lock sui dati tramite *AcquireWriterLock*, che prima di scrivere deve attendere che tutti i reader abbiano completato la lettura tramite *ReleaseReaderLock*. E' possibile promuovere un reader in un writer tramite *UpgradeToWriterLock* che restituisce un oggetto *LockCookie* necessario al metodo *DowngradeFromWriterLock*.

**Sincronizzazione con oggetti kernel di Windows** necessari per operazioni non possibili con gli altri metodi. Sono oggetti molto più complessi (circa 33 volte più esosi) che bisogna utilizzare solo se necessario. Tutti derivano da una classe comune *WaitHandle*, in cui il metodo *WaitOne* blocca il thread corrente fino alla segnalazione dell'oggetto kernel.

Grazie a questi oggetti kernel è possibile valicare i confini derivanti da processi e Application Domain. Per ottenere ciò viene creato il *WaitHandle* concreto specificandone un nome identificativo da cui è possibile riottenerlo da un'altro processo tramite il metodo statico WaitHandle.*OpenExisting* passando la stringa identificativa scelta. Se non viene trovato nulla con la stringa identificativa specificata, viene lanciata una *WaitHandleCannotBeOpenedException*.

**Mutex** consente sincronizzazioni come lock e Monitor, ma stavolta anche superando i confini derivanti da processi e da Application Domain. Si acquisisce tramite WaitHandle.WaitOne e si rilascia tramite Mutex.*ReleaseMutex*.

**Semaphore** consente l'accesso a una risorsa ad un insieme di thread. Nello specifico, un semaforo ha un insieme di slot validi (minimo e massimo specificabili nel costruttore). Quando gli slot sono tutti occupati il codice viene bloccato finquando un thread non rilascia il proprio slot tramite Semaphore.*Release*.

**Event** consente di notificare un evento a thread multipli (oltre i confini di processo e di Application Domain). Un evento può avere solo i due stati on/off. Esistono due tipi di eventi (che derivano dalla classe *EventWaitHandle*):

**AutoResetEvent** se l'evento viene notificato (on) viene automaticamente riportato nello stato disabilitato (off) dal primo oggetto in attesa dell'evento. Il comportamento risulta quindi simile al Mutex (simile al metodo Object.Notify di Java)

**ManualResetEvent** se l'evento viene notificato (on) tutti i thread in attesa dell'evento vengono sbloccati finquando qualcuno non riporta l'evento in uno stato disabilitato (simile al metodo Object.NotifyAll di Java)

## 7.4 Asynchronous Programming Model (APM)

APM consente di eseguire metodi di una classe in thread separati per migliorare la reattività dell'applicazione e non causare

il freeze, ad esempio, nella lettura di un file di testo molto lungo. Il metodo:

```
T       Method(P param)          throw Exception
```

se consente la sua esecuzione in un thread separato comprende le versioni specializzate:

```
IAsyncResult  BeginMethod(P param,
                  ASyncCallback c, object state)

T    EndMethod(IAsyncResult r)   throw Exception
```

I modelli rendezvous sono tre differenti modi per ottenere il risultato dell'operazione asincrona in APM:

**Modello Wait-Until-Done** utilizza EndXXX: blocca il thread corrente fino al termine dell'operazione

**Modello Polling** interroga ripetutamente *IAsyncResult.IsCompleted* per verificare il termine dell'operazione, lasciando ripetutamente il tempo di eseguire un'operazione (tipo un'indicatore di attesa)

**Modello Callback** viene creato il delegate *ASyncCallback* con il metodo da invocare al momento del completamento e opzionalmente con lo *stato* da passare

Eventuali eccezioni non sono sollevate nel momento in cui avviene l'errore, ma al momento dell'invocazione del metodo EndXXX.

I delegate sono molto utili in questo senso: comprendono i metodi BeginInvoke ed EndInvoke per abilitare qualunque metodo al modello APM.

### 7.5 ThreadPool

In molti casi crearsi i propri thread non è necessario: il threading system mantiene nativamente un gruppo di thread in background[13] riusabili il cui numero si adatta gradualmente alle esigenze. Per aggiungere un pezzo di lavoro a un thread si utilizza il metodo statico *ThreadPool.(Unsafe)QueueUserWorkItem*, che richiede un *WaitCallback* assieme al parametro del metodo. Il ThreadPool comprende due parametri:

**Numero di thread** numero di thread che fanno parte del pool (in un singolo processore 25)

**Completion Ports** speciali thread per operazioni I/O asincrone (in un singolo processore 1000)

Il ThreadPool fornisce il metodo *(Unsafe)RegisterWaitForSingleObject* per attendere l'avvio di un'operazione solo al momento della segnalazione di un oggetto WaitHandle.

Si possono ottenere migliori prestazioni utilizzando la versione *Unsafe* dei metodi, che non propaga lo stack della chiamata al thread di lavoro. Consente al codice di ignorare lo stack della chiamata e quindi migliorare la prestazioni, ma viceversa consente anche di aumentare i propri privilegi di protezione[14].

[13] i thread in background sono identici a quelli in foreground tranne per il fatto che i primi non mantengono attivo l'ambiente di esecuzione gestito. Dopo che *tutti* i thread in foreground sono stati interrotti tutti i thread in background vengono interrotti e viene effettuato un arresto del sistema

[14] lo stack del thread del pool di thread non disporrà del contesto degli effettivi chiamanti, e codice non autorizzato potrebbe riuscire a sfruttare questa situazione per evitare i controlli di autorizzazione

## 8 Application Domain

Un *Application Domain* è un contenitore logico che consente di eseguire molteplici assembly all'interno di un singolo processo ma impedendo allo stesso tempo che gli assembly che fanno parte dello stesso processo possano accedere direttamente alle aree di memoria non di loro appartenenza. Gli Application Domain posseggono dunque le stesse caratteristiche dei processi isolati, ma senza l'overhead necessario a creare un processo. In un'assembly è possibile creare vari Application Domain destinati ad eseguire vari assembly. Questo consente di invocare assembly riducendone i rischi non consentendo l'accesso alle risorse non consentite. Inoltre, se un assembly termina in modo anomalo, non viene terminato il processo, ma l'Application Domain a cui appartiene.

Nel Framework .NET la classe *System.AppDomain* consente di creare un Application Domain (metodo statico AppDomain.*CreateDomain*), di eseguire un'assembly su di esso (metodo istanza AppDomain.*ExecuteAssembly(ByName)*)[15] o Load, di rimuovere un'assembly (metodo statico AppDomain.*Unload*). Si può ottenere l'AppDomain corrente tramite il metodo statico *AppDomain.CurrentDomain* o *Thread.GetDomain* (sempre statico). La proprietà *AppDomain.ShadowCopyFiles* specifica se tutti gli assembly caricati nel dominio applicazione sono stati replicati. E' possibile limitare i privilegi di un'Application Domain (assembly) passando al metodo CreateAssembly (ExecuteAssembly) un oggetto *System.Security.Policy.Evidence*, che contiene le informazioni utili a determinare a quale code group l'assembly appartiene. Nella creazione dell'Evidence è necessario passare due array di tipo object: l'uno contenente l'host evidence (ad es. la Zone di appartenenza), l'altro l'assembly evidence. Infine è possibile controllare le opzioni di creazione di un AppDomain passando un *AppDomainSetup* al metodo CreateAssembly. Ad esempio, si può impostare la proprietà AppDomainSetup.*PrivateBinPath(Probe)*, che imposta l'elenco delle directory nella directory base dell'applicazione in cui effettuare il sondaggio degli assembly privati (*escludendo ApplicationBase*).

## 9 Servizi

I servizi consentono di eseguire assembly in background, senza interazioni utente, utili quando necessario monitorare in modo continuo qualcosa. Per crearli è necessario estendere la classe *ServiceBase* e implementare i metodi *OnStart* e *OnStop*.

Per installarli si può utilizzare lo strumento *InstallUtil.exe* o estendere un *Installer*. In quest'ultimo caso, per installare un servizio creare una classe di installazione che eredita da *Installer* e impostare l'oggetto *RunInstallerAttribute* della classe su true. All'interno del progetto, creare un'istanza di *ServiceProcessInstaller* per ogni applicazione di servizio (consente di installare un eseguibile contenente le classi che estendono ServiceBase) e un'istanza di *ServiceInstaller* per ogni servizio nell'applicazione. Infine, aggiungere l'istanza di ServiceProcessInstaller e le istanze di ServiceInstaller alla classe del programma di installazione del progetto. Ad esempio:

```
[RunInstallerAttribute(true)]
public class MyProjectInstaller: Installer {
   private ServiceInstaller serviceInstaller1;
   private ServiceInstaller serviceInstaller2;
   private ServiceProcessInstaller processInstaller;
```

[15] il primo carica da file, mentre il secondo carica da un nome di un assembly che appartiene al GAC

```
   public MyProjectInstaller(){
      // Instantiate installers for process and services.
      processInstaller = new ServiceProcessInstaller();
      serviceInstaller1 = new ServiceInstaller();
      serviceInstaller2 = new ServiceInstaller();

      // The services run under the system account.
      processInstaller.Account = ServiceAccount.LocalSystem;

      // The services are started manually.
      serviceInstaller1.StartType = ServiceStartMode.Manual;
      serviceInstaller2.StartType = ServiceStartMode.Manual;

      // ServiceName must equal those on ServiceBase derived classes.
      serviceInstaller1.ServiceName = "Hello-World Service 1";
      serviceInstaller2.ServiceName = "Hello-World Service 2";

      // Add installers to collection. Order is not important.
      Installers.Add(serviceInstaller1);
      Installers.Add(serviceInstaller2);
      Installers.Add(processInstaller);
   }
}
```

La proprietà ServiceProcessInstaller.*Account* consente di specificare il contesto di sicurezza:

**LocalService** nel computer locale: utente non privilegiato. In rete: credenziali anonime

**NetworkService** nel computer locale: utente non privilegiato. In rete: credenziali del computer

**LocalSystem** nel computer locale: utente privilegiato. In rete: credenziali del computer

**User (default)** quando il servizio è installato viene richiesto un nome utente valido e relativa password

Per controllare i servizi si può utilizzare il comando NET o la classe ServiceController, come da esempio:

```
ServiceController sc = new ServiceController("WindowsUpdate");
sc.MachineName = "pc01";
sc.Stop();
Thread.Sleep(2000);
sc.Start();
```

E' possibile avviare e interrompere i servizi anche utilizzando i metodi statici *ServiceBase.Run(ServiceBase)*: l'unica differenza sta che mentre il primo vuole il nome del servizio, il secondo vuole un'istanza di ServiceBase.

# 10 Configurare le applicazioni

Il namespace *System.Configuration* comprende tutte le classi utili a gestire la configurazione di un'applicazione.

I file setting hanno lo scopo di facilitare i parametri di avvio dell'applicazione. Visual Studio assiste l'inserimento di variabili di qualunque tipo e crea una classe wrapper che deriva da *ApplicationSettingsBase* che consente di ottenere e reimpostare tali valori in modo fortemente tipizzato. Le variabili possono avere due tipi di scope: Application o User.

A volte è però necessario maggior controllo sulle configurazioni: si vorrebbe ad esempio intervenire direttamente sul file di configurazione per cambiare la stringa di connessione. Premesso che si andrebbe incontro a problemi di sicurezza, accedere alle impostazioni più a basso livello è possibile, naturalmente a discapito di una maggiore complessità (compensato da una totale libertà di utilizzo). La classe *ConfigurationManager*[16] consente di creare istanze Configuration tramite metodi *Open(Mapped)(Exe/Machine)Configuration*. La versione "Machine" apre il file `Machine.config`. La versione "Mapped" consente di specificare manualmente il file di configurazione. La versione "Exe" apre una configurazione client, che consente di specificare una enumeration *ConfigurationUserLevel* per aprire una configurazione comune applicata a tutti gli utenti (`nomeApplicazione.exe.config`) o solo per l'utente corrente in un contesto di *roaming*[17] o di computer locale.

Un file di configurazione è un file xml formato dai seguenti elementi:

**startup** come l'applicazione verrà avviata

- **supportedRuntime** se (e solo se) la versione di .NET con cui è stata compilata l'applicazione non esiste nella macchina di esecuzione, forza l'utilizzo della versione indicata. Se non ci fosse stata tale indicazione sarebbe stata eseguita con la versione più recente di .NET disponibile sulla macchina. Quando sono supportate più versioni di CLR, il primo elemento deve indicare la versione preferita, mentre l'ultimo elemento quella meno desiderata

**runtime** come l'applicazione verrà eseguita

- **developmentMode** consente di installare l'assembly nel Global Assembly Cache (GAC). Risulta necessario creare una variabile di ambiente `DEVPATH` e impostare l'attributo *developerInstallation* a true. Inserire l'elemento al file "machine.config" nel computer di sviluppo (è quindi un'impostazione globale, non locale dell'applicazione)
- **assemblyBinding** consente di assegnare una path all'assembly utilizzando lo specifico tag "codebase" che consente di specificare sia la localizzazione che la versione di un assembly
  - **dependentAssembly/bindingRedirect** un'applicazione può essere formata da componenti che utilizzano versioni diverse di .NET Framework: questa situazione può generare conflitti tra le dipendenze dei componenti. Per risolvere questo problema, quando un'applicazione utilizza componenti creati con versioni diverse di .NET Framework, *la versione del runtime associato all'applicazione determina la versione degli assembly* di .NET Framework che verranno utilizzati dall'applicazione e da tutti i relativi componenti.
    È possibile annullare questa funzionalità aggiungendo elementi *bindingRedirect* ai file Machine.config o Web.config. In tal modo un'applicazione può utilizzare una versione aggiornata di assembly progettata per sostituire un assembly esistente per alcuni tipi di applicazioni.

**appSettings** liste chiavi/valore tramite elemento "add". I valori sono ottenibili tramite il metodo statico *ConfigurationManager.AppSettings* che li restituisce come mappa *NameValueCollection*

[16] la vecchia classe ConfigurationSettings esiste ancora solo per motivi di compatibilità

[17] possibilità di portare automaticamente la configurazione di un utente da un computer ad un altro. "Documents and Settings/[Nome utente]/Dati applicazioni/[ AssemblyCompany]/[Nome applicazione]/[ AssemblyVersion]/user.config" rappresenta le impostazioni dell'utente in un contesto di roaming

**connectionStrings** modo sicuro ed elegante per memorizzare stringhe di connessione al database tramite elemento "add". Contiene:

**clear** elimina ogni stringa di connessione preesistente

**name** per referenziare tramite nome senza bisogno di ricordarsi l'indice

**providerName** ad esempio System.Data.OracleClient

**connectionString** parametri di connessione

I valori sono ottenibili tramite il metodo statico *ConfigurationManager.ConnectionStrings* che li restituisce come lista *ConnectionStringSettingsCollection* tipizzato fortemente

**configSections** consente di specificare sezioni di configurazione xml *sectionGroup*, restituibili tramite ConfigurationManager.*GetSection*. Visual Studio consente di creare una sezione in modalità fortemente tipizzata creando una classe che estende *ConfigurationSection*[18]. Per ottenere il valore xml della sezione utilizzare ConfigurationSection.SectionInformation.GetRawXml.

La proprietà Configuration.*SectionGroups* restituisce la lista *ConfigurationSectionGroupCollection*. L'attributo *ConfigurationProperty* consente di specificare se una proprietà è obbligatoria e il suo valore di default.

**system.runtime.remoting** impostazioni di .NET remoting

**application/service/wellknown** contiene informazioni relative a oggetti (noti) attivati da server ed esposti ai client dall'applicazione. Prevede i seguenti attributi:

**mode** Singleton o SingleCall. Con Singleton esisterà sempre una sola istanza, indipendentemente dal numero di client presenti per ciascun oggetto, mentre con SingleCall un nuovo oggetto viene creato dal sistema per ciascun metodo client richiamato

**type** specifica il nome completo del tipo dell'oggetto e il nome dell'assembly contenente l'implementazione del tipo

**objectUrl** Specifica l'endpoint dell'URI dell'oggetto. Quando un oggetto è contenuto in Internet Information Services (IIS), l'estensione objectUri deve essere ".soap" o ".rem", in modo che la richiesta venga inviata all'IHttpHandler di .NET Framework Remoting.

**system.diagnostics** impostazioni di tracing/debug

**switches** consente di aggiungere switch, ottenibili tramite il costruttore di *BooleanSwitch*. Utile ad esempio per abilitare o meno il trace su console

Per salvare le modifiche di una configurazione si utilizza *Configuration.Save(As)*, passando opzionalmente anche un ConfigurationSaveMode che consente di scegliere tra:

**Full** tutte le proprietà sono salvate

**Modified** tutte le proprietà modificate sono salvate, anche se i valori modificati sono uguali a quelli ereditati

**Minimal** tutte le proprietà che differiscono dalle proprietà ereditate sono salvate

# 11 Creare un Installer

Il principale strumento per installare applicazioni è la classe base *Installer*, che richiede di:

1. ereditare la classe Installer
2. sovrascrivere i metodi Install, Commit, Rollback, Uninstall (le installazioni sono transazionali)
   - la sequenza dei metodi è la seguente: OnBefore(Un)Install, (Un)Install, OnAfter(Un)Install, OnCommitting, Commit, OnCommitted o in alternativa OnBeforeRollback, Rollback, OnAfterRollback
3. opzionalmente aggiungere nel costruttore i metodi delegate da eseguire al momento degli eventi Committing, Committed, ...
4. aggiungere alla classe l'attributo RunInstaller e impostarlo a true
5. invocare l'installer tramite InstallUtil.exe

Implementazioni pronte all'uso sono *AssemblyInstaller*, che carica un assembly ed esegue tutti gli installer in esso contenuti o *ComponentInstaller*, che specifica un installer che consente di copiare le proprietà da un componente da utilizzare in fase di installazione.

# 12 Strumentazione

## 12.1 Log di eventi

Il sistema "Windows Event and Logging" consente di registrare quelle informazioni utili allo sviluppatore per correggere eventuali bug (non supportate in versioni inferiori al Windows ME compreso). Non si deve utilizzare né in presenza di partial trust environment né in modo troppo intensivo per non compromettere le prestazioni dell'intero sistema. Il log di eventi è l'unico modo tramite il quale un servizio può comunicare con l'utente[19].

Per creare un log di eventi, si usa nel namespace *System.Diagnostics* la classe *EventLog*. Richiede per l'inizializzazione l'EventLog.*Log* (nativi: Application, Security[20], System) e l'EventLog.*Source* e per la scrittura il metodo EventLog.*WriteEntry* in cui è possibile specificare l'*EventLogEntryType* e un event ID EventLog.InstanceId. Si possono aggiungere ulteriori informazioni passando al costruttore di EventLog un EventSourceCreationData.

E' possibile creare nuovi fonti EventLog, ma questo richiede privilegi amministrativi. Per questo motivo tale operazione viene normalmente eseguita nella fase di installazione.

[18] il precedente modo, deprecato, prevedeva l'implementazione dell'interfaccia *IConfigurationSectionHandler* in modo thread safe e privo di stato

[19] la comunicazione sarebbe possibile anche tramite file di testo, ma non è conveniente per applicazioni aziendali

[20] il log Security è di sola lettura: può infatti essere scritto solo dal sistema operativo per riportare eventi di audit

Per iterare gli eventi di un log (*ma non di un source*[21]) si analizza la proprietà EventLog.*Entries* che restituisce un *EventLogEntryCollection* e analizzare ogni proprietà *EventLogEntry.Message*. Per pulire un log si utilizza il metodo EventLog.*Clear* mentre per cancellarlo si usa il metodo statico EventLog.*Delete*.

## 12.2 Debugging e Tracing

Poiché procedere step per step alla ricerca di un errore è troppo dispendioso si usano le classi Debug e Debugger in modo che il codice stesso fornisca un riscontro sulla correttezza del programma.

La classe *Debugger* consente di inserire breakpoint tramite Debugger.*Break*. Tramite questo approccio il breakpoint non rimane vincolato alle impostazioni del progetto ed è possibile inserire breakpoint condizionali. Il metodo Debugger.*Log* invia informazioni al debugger collegato passando livello, categoria e messaggio.

Se è necessaria maggiore granularità bisogna utilizzare la classe Debug. In particolare, il metodo Assert consente di verificare condizioni e in caso di fallimento visualizza una dialog con un messaggio e consente il debug. Il metodo Fail è analogo ad Assert(false). I metodi Write(Line)(If) stampa un messaggio nell'output di Visual Studio, mentre in Print - Flush l'output è rappresentato dai listener registrati. Il metodo Assert in versione release non viene valutato: questo non significa solo che l'implementazione di Assert non restituisce nulla, ma anche che i suoi argomenti non vengono valutati. Quindi non è conveniente inserire:

```
bool result = DoComplexCheck();
Debug.Assert(result, "Check fallito");
```

Infatti, a prescindere dal tipo di compilazione il metodo DoComplexCheck viene comunque valutato. Se invece si considera:

```
Debug.Assert(DoComplexCheck(), "Check fallito");
```

Se la compilazione è in versione release, allora il metodo DoComplexCheck stavolta non verrà valutato.

Per controllare la modalità di esecuzione del debugging è possibile utilizzare gli attributi:

**DebuggerBrowsable** come mostrare l'elemento (Never, Collapsed, RootHidden)

**DebuggerDisplay** il ToString ad uso esclusivo del debugger

**DebuggerHidden** su qualunque tipo di codice (esterno/interno) non considera breakpoint e non viene mostrato nel callstack

**DebuggerNonUserCode** su codice esterno[22] non considera breakpoint (warn) e non viene mostrato nel callstack. Se interessa una singola proprietà è possibile mostrarla con l'attributo *DebuggerStepperBoundary*

[21] se si vuole analizzare se il source dell'evento correntemente iterato corrisponde a uno specifico source diventa quindi necessario analizzare la relativa proprietà EventLog.Source

[22] solo se impostata l'opzione "Tools/Options/Debugging/Enable Just My Code" su OFF

| | Interno | Esterno |
|---|---|---|
| **Breakpoint *OFF*** | Hidden | Hidden |
| | | NonUserCode |
| | StepThrough | StepThrough |
| **Callstack *OFF*** | Hidden | Hidden |
| | | NonUserCode |
| | | StepThrough |

Tabella 2: Relazioni tra Hidden, StepThrough e NonUserCode

**DebuggerStepThrough** su codice esterno non viene mostrato nel callstack, mentre indipendentemente dal tipo di codice non considera breakpoint

**DebuggerTypeProxy** rimpiazza completamente la visualizzazione dell'oggetto con un suo rappresentante (proxy)

**DebuggerVisualizer** associa un visualizzatore per la classe (ad esempio il visualizzatore per la classe Color è la tavolozza dei colori)

La tab. 2 sintetizza gli attributi appena illustrati.

La classe *Trace* è simile alla classe Debug, ma mentre quest'ultima funziona solo in modalità debug, Trace funziona sia in modalità Debug che Release. Per abilitare il tracing si utilizza TraceSource.*TraceInformation*, per controllarlo si usa TraceSource.*Switch*. TraceLevelSwitch prevede i seguenti valori enum: Off, Error, Warning, Info e Verbose.

Sia Trace che Debug possiedono una proprietà Listeners di tipo TraceListenerCollection. Implementazioni di TraceListener sono: *Default*TraceListener (sulla console di VisualStudio), *Console*TraceListener (su stdout), *TextWriter*TraceListener (su stream), *XmlWriter*TraceListener, *EventLog*TraceListener (su event log), *DelimitedList*TraceListener (con carattere di separazione per import in fogli di calcolo).

La classe *CorrelationManager* consente di mantenere le richieste di Trace o di Debug isolate e uniche tra differenti processi.

## 12.3 Monitorare prestazioni

### 12.3.1 Processi

Un'istanza della classe *Process* coincide con uno o più processi in esecuzione sulla macchina locale o remota. Per ottenere i processi è possibile utilizzare i metodi statici Process.GetCurrentProcess, GetProcessBy(Id/Name), GetProcesses (nel caso di macchina remota si passa anche il nome della macchina). Per avviarli si passa un ProcessStartInfo al metodo statico *Process.Start*, aggiungendo opzionalmente l'applicazione o il documento da avviare (proprietà ProcessStartInfo.FileName). A ProcessStartInfo è possibile anche omettere l'applicazione da eseguire dichiarando al suo posto il file da aprire, che verrà aperto con l'applicazione di default registrata in Windows. Quindi è conveniente utilizzare ad esempio:

```
ProcessStartInfo p = new ProcessStartInfo("index.html")
```

rispetto a:

```
ProcessStartInfo p = new ProcessStartInfo("iexplore.exe
p.FileName = "index.html";
```

Mentre il secondo infatti forza ad utilizzare Internet Explorer per l'apertura di un sito web, il primo consente di aprire la pagina utilizzando il browser di default (Firefox, Opera, ...).

Per terminare un processo si utilizza il metodo d'istanza *Process.Kill*.

### 12.3.2 Contatori

Per correggere errori di prestazioni è necessario aver calcolato una misurazione delle prestazioni minime accettabili e confrontarle con i valori prodotti. Per eseguire questo confronto è necessaria la classe *PerformanceCounter*, abilitandone il corrispondente PerformanceCounterPermission solo al codice full trust. Tale classe consente di memorizzare dati nel registro di sistema in modo da essere *globalmente accessibili* e le proprietà che lo identificano[23] sono: nome computer, nome categoria, istanza categoria, nome counter (ad esempio "localhost, Processor, % Processor Time, _Total"). Altre informazioni si possono trasferire passando al costruttore un'istanza *CounterCreationData*. Le categorie coincidono con *PerformanceCounterCategory*, che contiene vari *InstanceDataCollection*. Per creare nuove categorie si utilizza il metodo statico PerformanceCounterCategory.Create, previo controllo statico in PerformanceCounterCategory.Exists (altrimenti eccezione). Si possono le seguenti categorie (PerformanceCounterCategoryType):

**MultiInstance** la categoria del contatore delle prestazioni può avere più istanze.

**SingleInstance** la categoria del contatore delle prestazioni può avere solo un'istanza singola (Singleton).

**Unknown** la funzionalità dell'istanza per la categoria del contatore delle prestazioni non è nota

Per modificare il valore del PerformanceCounter personalizzato chiamare i metodi *Increment(By)*, *Decrement(By)* o impostare la proprietà *RawValue*. Tali metodi sono invocabili solo se la proprietà *PerformanceCounter.ReadOnly* è impostata a false poiché di default un PerformanceCounter è di sola lettura. I metodi Increment, IncrementBy e Decrement utilizzano i blocchi per l'aggiornamento del valore del contatore. Ciò fa in modo che il valore del contatore sia *accurato negli scenari multithread o multiprocesso* ma implica anche una riduzione di prestazioni. Se la precisione fornita dalle operazioni con blocchi non è necessaria, è possibile aggiornare la proprietà *RawValue direttamente per ottenere un miglioramento delle prestazioni* fino a 5 volte superiore. Tuttavia, negli scenari multithread, alcuni aggiornamenti al valore del contatore potrebbero essere ignorate, fornendo dati non accurati.

Lo *StackTrace* consente di ispezionare in ogni momento lo stato dell'applicazione analizzando gli elementi StackFrame che lo compongono.

## 12.4 Rilevare Management Event

Per interagire e modificare le informazioni di sistema si utilizzano un insieme di interfacce note con il nome di Windows Management Instrumentation (WMI), accessibili tramite System.Management.ManagementObjectSearcher secondo una sintassi sql-like. Per eseguire una query è necessario:

1. se si vuole interrogare una risorsa non locale, ma remota:
   (a) creare un'istanza ConnectionOptions impostando le proprietà UserName e Password
   (b) creare un'istanza ManagementScope passando il PathName e le ConnectionOptions. Il PathName è generalmente espresso nella forma: "*MachineName NameSpace*", dove esistono i seguenti namespace principali:
   **root/cimv2** Win32, PerformanceCounter, Event Log, Windows Installer, Sicurezza
   **root/snmp** SNMP
   **root/default** Registry
   **root/wmi** WDM
   **root/directoryldap** Directory Services
   **root/*** Views
2. creare un'istanza *ObjectQuery* passando la query[24]
3. creare un'istanza ManagementObjectSearcher passando il ManagementScope (se necessario) e l'ObjectQuery
4. scandire il set di tuple ManagementObjectCollection restituito da ManagementObjectSearcher.Get. Ogni valore della singola tupla ManagementObject è restituibile tramite l'indexer specificando come chiave il nome della colonna.

La query utilizza una sintassi simile all'sql e consente di interrogare tabelle di sistema come Win32_LogicalDisk, Win32_NetworkConfiguration, Win32_Service.

Per ricevere notifiche su aggiornamenti nei risultati di una query si può passare un *EventQuery* (tipicamente in sintassi WQL[25]) a un ManagementEventWatcher invocando il metodo ManagementEventWatcher.WaitForNextEvent per monitoraggi sincroni. In caso invece di monitoraggi asincroni bisogna aggiungere un handler all'evento ManagementEventWatcher.EventArrived e avviare il metodo ManagementEventWatcher.Start. Quando si vuole interrompere le notifiche (sia sincrone che asincrone) si invoca il metodo ManagementEventWatcher.Stop.

# 13 Sicurezza delle applicazioni

## 13.1 Code Access Security (CAS)

Code access security (CAS) consente di controllare i permessi di un'applicazione a un livello molto granulare, come il filesystem, il registro, le stampanti, gli event log. Purtroppo CAS può essere applicato solo alle applicazioni managed, mentre quelle unmanaged hanno solo il limite imposto dal sistema operativo tramite la Role Based Security (RBS). Le applicazioni fully trusted sono esentate dal controllo CAS (tra cui quindi anche le applicazioni unmanaged).

Come nei sistemi operativi si usa l'username per identificare l'utenza, in .NET si usano le *Evidence* per identificare gli assembly. Gli assembly sono identificabili in vari modi[26] tra cui

[23] l'identificazione può essere paragonata al check di un path

[24] in caso di semplici select, si può utilizzare la *SelectQuery* (sottoclasse di ObjectQuery), in cui bisogna solo dichiarare la tabella da scandire

[25] WQL è un sottoinsieme dell'SQL espressamente progettato per WMI. Comprende clausole speciali come WITHIN per l'intervallo di polling, o ISA per confronti lessicografici

[26] si potrebbe leggere quindi come: l'Evidence può essere creato in vari modi

tramite location o hash o signature dell'assembly. Esistono due tipi di evidence:

**Host evidence** origine dell'assembly

**Assembly evidence** identificazione custom

Una volta identificato un assembly è possibile assegnare dei permessi tramite *System.Security.Permission* sia in modo dichiarativo che imperativo. I *Permission Set* raccolgono differenti permessi secondo regole ben definite. Il framework .NET ne comprende sette di default, tra cui Internet[27], LocalInternet, FullTrust, Execute[28] . Il mapping Evidence-PermissionSet (quindi i permessi consentiti da un assembly) è il *CodeGroup*. Come un account utente può essere membro di diversi gruppi, un assembly può essere un membro di più permission set (unione dei permessi[29]). Inoltre, si possono annidare i code group (intersezione dei permessi[30]), modificabile tramite l'attributo *LevelFinal*, che esclude la valutazione dei livelli di criteri inferiori al livello corrente. La *Security Policy* rappresenta un raggruppamento logico di code group e permission set: esistono di default le policy Enterprise, Machine, User, Application Domain. Un assembly può quindi infine far parte di più policy e in questo caso viene calcolata l'intersezione dei permessi[31]. Per visualizzare i permessi richiesti, concessi e negati di un'applicazione si usa il .NET Framework Configuration Tool o l'applicazione *permview.exe*.

CAS è completamente indipendente dalla sicurezza del sistema operativo e lavora in cima ad esso. Si può esaminare e modificare le policy CAS sia tramite il .NET Framework Configuration Tool sia tramite l'applicazione `Caspol.exe`.

*ApplicationSecurityInfo* fornisce informazioni di protezione su un'applicazione attivata da un manifest utilizzando le informazioni ottenute dai manifesti dell'applicazione e dalla relativa classe *ActivationContext*.

## 13.2 Sicurezza dichiarativa

A prescindere dalle policy (che cambiano da sistema a sistema), la sicurezza dichiarativa è utile per verificare che l'*assembly* che si sta sviluppando abbia tutti i permessi necessari (ma nessuno in più) gestendo opportunamente eventuali errori.

CAS consente di restringere i permessi necessari a differenti tipi di risorse per le quali esiste una corrispondente classe che deriva da *CodeAccessPermission*. Poiché tutti i permessi derivano dalla stessa classe base, tutte condividono due proprietà standard:

**Unrestricted** abilita tutti i permessi della classe corrente

**Action** l'azione di sicurezza da prendere, a scelta nell'enum SecurityAction, che comprende 9 valori di cui 3 per l'assembly e 6 per i metodi. I 3 dell'assembly sono:

**RequestMinimum** [32] permessi obbligatori richiesti, altrimenti *PolicyException*

**RequestOptional** [33] rifiuta tutti i permessi eccetto quelli elencati in RequestOptional o RequestMinimum, ma non verifica se tali permessi esistono (usare per questo RequestMinimum)

**RequestRefuse** rifiuta il permesso specificato, ma non verifica se è riuscito effettivamente a rifiutarlo (usare per questo RequestMinimum). Utile per raffinare i permessi di una RequestOptional.

In modo dichiarativo è quindi possibile vincolare metodi e classi. Per vincolare l'intero assembly anteporre "assembly:" come ad esempio:

```
[assembly: FileIOPermissionAttribute(
           SecurityAction.RequestOptional, Read = @"C:\")]
[assembly: FileIOPermissionAttribute(
           SecurityAction.RequestRefuse, Read = @"C:\Windows\")]
```

## 13.3 Sicurezza dichiarativa e imperativa per metodi

Si può utilizzare il CAS anche per proteggere *metodi* individuali e per questo è necessario utilizzare sia la tecnica dichiarativa che imperativa. Come detto precedentemente, l'enum SecurityAction prevede 6 valori per i metodi:

**(Revert)Assert** chiede al runtime di ignorare se mancano i permessi specificati (deve essere a sua volta permesso poterli ignorare[34]). Utile per consentire permessi speciali in codice partially trusted

**(Revert)PermitOnly** nega *tutti* i permessi tranne quello specificato. Analogo a RequestOptional dell'assembly

**Deny** nega *un* permesso. Utile per raffinare PermitOnly. Analogo a RequestRefuse dell'assembly

**Demand** lancia un eccezione se *tutti* i chiamanti dello stack non hanno il permesso specificato. Utile per forzare il controllo di sicurezza in codice unmanaged

**LinkDemand** lancia un eccezione se *il solo chiamante in cima* allo stack non ha il permesso specificato. E' più rapido rispetto a Demand, ma più permissivo. Utile ad esempio se voglio permettere la connessione a internet solo all'assembly dotato di una chiave pubblica specifica. Tale assembly sarà quello che invoca direttamente tale metodo, non gli altri nello stack

**InheritanceDemand** lancia un eccezione se la classe/metodo che deriva dalla classe/metodo annotato non possiede il permesso specificato

Il Revert anteposto ai metodi (presente come metodi statici in CodeAccessPermission) è utile nel caso imperativo per ripristinare la sicurezza. Per verificare i permessi consentiti utilizzare il metodo *SecurityManager.IsGranted*.

Si può infine creare un *PermissionSet* e utilizzare il metodo AddPermission per aggiungere vari CodeAccessPermission.

[27] in cui è negato inviare richieste a siti web

[28] fornisce i privilegi più bassi pur garantendo l'esecuzione dell'assembly

[29] viene dato il massimo privilegio. Come dire: l'utente root appartiene ai gruppi user e root e assegno i privilegi di root

[30] minimi privilegi

[31] se un'applicazione per l'utente ha permessi di rete ma per la società no l'applicazione non potrà accedere alla rete

[32] sarebbe stato meglio "RequireMinimum"

[33] sarebbe stato meglio "RefuseAllExcept"

[34] deve essere consentito il privilegio SecurityPermissionFlag.Assertion, garantito di default ai permission set FullTrust, LocalInternet e Everything

Per realizzare una logica di sicurezza personalizzata, finalizzata a determinare se un'applicazione .NET possa essere eseguita o meno è necessario implementare l'interfaccia IApplicationTrustManager, che consente di determinare se un'applicazione deve essere eseguita e con quale insieme di autorizzazioni. L'host chiama il metodo *DetermineApplicationTrust* del gestore di attendibilità per determinare se è necessario eseguire un'applicazione e quali autorizzazioni devono essere concesse all'applicazione.

# 14 Sicurezza delle utenze e dei dati

Le imprese devono poter controllare l'accesso a dati confidenziali. La protezione dei dati necessita il coordinamento di diverse tecnologie:

**Role Based Security (RBS)** in base all'utente e al gruppo consente di stabilire quali operazioni sono consentite nell'organizzazione

**Access Control List (ACL)** in base all'utente e al gruppo consente di stabilire i diritti di accesso ai dati (rwx)

**Crittografia** consente di criptare, convalidare e firmare i dati

## 14.1 Autenticare e autorizzare gli utenti

L'*autenticazione* è il processo di identificazione dell'utente mentre l'*autorizzazione* è il processo di verifica dei permessi dell'utente. L'autorizzazione in genere si svolge solo dopo l'autenticazione.

La classe *System.Security.Principal.WindowsIdentity* rappresenta un account utente Windows. Non consente di autenticare l'utente ma memorizza semplicemente i risultati dell'autenticazione eseguita da Windows. Per creare un istanza è possibile utilizzare i metodi statici di WindowsIdentity *GetAnonymous*, *GetCurrent* e *Impersonate* (come quando si usa il comando "Esegui come"). Tramite l'istanza ottenuta si possono quindi ottenere informazioni sui dati dell'autenticazione come username, token, tipo di autenticazione, ... Per ottenere informazioni specifiche su un'account (quindi non su uno specifico utente o gruppo) si usa NTAccount che possiede metodi per ottenere ad esempio il SID di un'utente (security id).

La classe WindowsPrincipal fornisce accesso all'appartenenza di un utente a un gruppo. E' possibile creare una WindowsPrincipal specificando nel costruttore la WindowsIdentity. Modi alternativi è tramite Thread.CurrentPrincipal, non prima di aver impostato AppDomain.CurrentDomain.*SetPrincipalPolicy* a WindowsPrincipal in modo da consentire il cast esplicito (altrimenti restituisce una GenericPrincipal, senza informazioni associate). Grazie a WindowsPrincipal è possibile verificare l'appartenenza di un'utente a un gruppo tramite WindowsPrincipal.*IsInRole* dando un enum WindowsBuiltInRole. Se non si conosce il nome del computer è possibile utilizzare System.Environment.*MachineName* (solo locale) o System.Environment.*UserDomainName* (locale o Active Directory).

La classe System.Security.Permissions.*PrincipalPermission* verifica i permessi del IPrincipal attivo (passato nel costruttore): esegue dichiaratamente o imperativamente il PrincipalPermission.Demand verificando la correttezza delle seguenti proprietà impostabili[35]:

**Name** nome utente dell'identità da verificarne la corrispondenza

**Role** ruolo dell'identità da verificarne la corrispondenza

**Authenticated** richiede che l'utente sia autenticato

Esempio dichiarativo[36]:

```
[PrincipalPermission(SecurityAction.Demand,
      Name = @"CONTOSO\User1", Role = @"BUILTIN\Administrators")]
[PrincipalPermission(SecurityAction.Demand,
      Authenticated = true)]
static void AdministratorsOnlyMethod() { /* ... */ }
```

Se le funzionalità base di GenericIdentity e GenericPrincipal non sono sufficienti, per creare una identity o una principal personalizzata è necessario implementare l'interfaccia *IIdentity* o *IPrincipal*. Per la prima è necessario implementare le proprietà AuthenticationType, Name, IsAuthenticated mentre nella seconda bisogna implementare il metodo IsInRole. Implementare le interfacce abilita l'uso del demand sia dichiarativo che imperativo.

Le eccezioni che possono essere sollevate in un demand sono:

**AuthenticationException** necessario chiedere credenziali differenti e riprovare

**InvalidCredentialException** lo stream di autenticazione si trova in uno stato invalido e non è possibile riprovare (attenzione: controintuitivo!)

## 14.2 Utilizzare le ACL

ACL è la tecnica più comune per restringere l'accesso a file, cartelle, servizi, registro, chiavi crittografiche, handle EventWait, mutex e semafori.

Un *Discretionary Access Control List* (*DACL*) è un meccanismo di autorizzazione che individua quali utenti e gruppi hanno accesso a un oggetto. Se un DACL non identifica esplicitamente un utente o un gruppo l'accesso è negato. Di base un DACL è controllato dal possessore o dal creatore dell'oggetto e contiene tutti gli Access Control Entry (ACE). Per gestire in modo più efficiente l'ACL riducendo il numero di ACE necessari si utilizza il concetto di ereditarietà: ogni nuova cartella creata in `C:` eredita i permessi assegnati in `C:`

.

Il calcolo dei permessi non è banale poiché un ACL contiene vari ACE che possono assegnare privilegi direttamente all'utente o a un gruppo. Inoltre un utente può far parte di più gruppi e i gruppi possono essere annidati. I permessi concessi sono cumulativi, creando l'unione dei permessi, ma i permessi esplicitamente negati hanno la precedenza. Se non esiste alcun ACE applicabile all'utente analizzato di base il permesso è sempre negato.

Un *Security Access Control List* (*SACL*) determina se un evento (apertura file/cartelle) deve essere registrato (audit) nel security log per identificare intrusioni.

Per specificare i permessi dei file e delle cartelle si usa l'enum *FileSystemRights*. Le classi nel namespace System.Security.AccessControl consentono di configurare DACL,

[35] l'ordine seguente viene utilizzato nella forma imperativa

[36] l'esempio funziona anche con Role = @ "Administrators"

SACL, ACE. Per ogni tipo di risorsa (NativeObject, Directory, File, Registry, Mutex) esistono le seguenti classi ACL:

**<Type>Security** metodi statici per ottenere gli ACE di un DACL (GetAccessRules) o gli ACE di un SACL (GetAuditRules) e aggiungere o rimuovere ACL. Restituiscono in ordine una collezione di <Type>AccessRule o di <Type>AuditRule. Se si usa invece l'istanza restituita da <Type>.GetAccessControl o creata specificandone il file è possibile inserire nuovi ACE tramite AddAccessRule o configurarli tramite SetAccessRuleProtection[37] e reimpostarlo tramite <Type>.SetAccessControl.

**<Type>AccessRule** insieme di DACE. Eredita da AccessRule che a sua volta eredita da AuthorizationRule

**<Type>AuditRule** insieme di SACE. Eredita da AuditRule che a sua volta eredita da AuthorizationRule

## 14.3 Cifrare e decifrare dati

Il namespace System.Security.Cryptography consente di cifrare e decifrare i dati.

### 14.3.1 Crittografia Simmetrica

Nella *crittografia simmetrica* sono supportati i seguenti algoritmi di cifratura:

**RijndaelManaged** l'unico algoritmo che non sia unmanaged code. Anche conosciuto come *AES*[38]. Può utilizzare lunghezze di chiavi che vanno da un minimo di 128 bit a un massimo di 256, a incrementi di 32 bit

**DES** utilizza solo 56 bit ed è attualmente molto vulnerabile agli attacchi

**RC2** rimpiazzo del DES che utilizza dimensione delle chiavi variabile

**TripleDES** applica tre volte l'algoritmo DES (da 128 bit a 196 bit)

Bisogna utilizzare l'algoritmo Rijndael se sia il cifratore che il decifratore sono eseguiti su Windows XP o successivi; altrimenti utilizzare TripleDES.

Tutti gli algoritmi simmetrici derivano dalla classe base SymmetricAlgorithm e condividono le seguenti proprietà:

**BlockSize** dimension unità minima da crittografare in bit

**KeySize** dimensione chiave segreta in bit

**Key** la chiave. Se non specificata viene automaticamente generata. Dopo la cifratura è necessario trasferire in una modalità sicura tale chiave al decryptor in modo che la decifratura possa utilizzare la stessa chiave

**IV** Initialization Vector: dati utilizzati per oscurare ulteriormente il primo BlockSize da cifrare e che deve essere lo stesso sia per l'encryptor che per il decryptor. Per eliminare l'overhead richiesto nel trasferire tramite un canale sicuro l'IV tra encryptor e decryptor si potrebbe definire l'IV staticamente o derivarlo dalla Key

**Mode** modalità di funzionamento dell'algoritmo simmetrico. Il valore predefinito è CipherMode.CBC.

e i seguenti metodi:

**GenerateIV** genera un valore random di IV. Non è necessario utilizzare questo metodo poiché viene automaticamente generato ma risulta utile se si vuole utilizzare un nuovo IV

**GenerateKey** genera un valore random di Key. Come prima, non è necessario utilizzare questo metodo poiché viene automaticamente generato ma risulta utile se si vuole utilizzare una nuova Key

**ValidKeySize** utile per verificare se la lunghezza di una chiave è valida per l'algoritmo in uso

**CreateEncryptor** crea un oggetto ICryptoTransform utilizzato da CryptoStream che avvolge lo stream di output per abilitare la cifratura. Dato un array di byte di input, scrivera sullo stream gli stessi byte cifrati tramite il metodo Write

**CreateDecryptor** crea un oggetto ICryptoTransform utilizzato da CryptoStream per avvolge lo stream di input per abilitare la decifratura. Dato un array di byte di output, leggerà dallo stream gli stessi byte decifrati tramite il metodo Read

Per stabilire una chiave simmetrica si può utilizzare la classe *Rfc2898DeriveBytes* per trasformare una password in una key. Per questo è necessario condividere tra encryptor e decryptor i seguenti valori: password, salt value, IV e numero di iterazioni. La chiave calcolata viene restituita tramite Rfc2898DeriveBytes.GetBytes, che richiede il numero di byte: per la Key bisogna specificare KeySize/8, mentre per l'IV bisogna specificare BlockSize/8. In alternativa, è possibile utilizzare RandomNumberGenerator.GetBytes che consente di riempire la matrice di byte *passata per parametro* con una sequenza di valori casuali, in cui la matrice deve essere lunga KeySize/8 per la Key o BlockSize/8 per l'IV.

### 14.3.2 Crittografia Asimmetrica

Nella *crittografia asimmetrica* sono supportati i seguenti algoritmi di cifratura:

**RSACryptoServiceProvider** implementazione dell'algoritmo RSA. Rappresenta il wrapper managed sulla implementazione unmanaged di RSA

**DSACryptoServiceProvider** Utilizzato per il firmare elettronicamente i messaggi.

RSACryptoServiceProvider fornisce i seguenti metodi/proprietà:

[37] prevede due parametri:

**isProtected** per proteggere le regole di accesso dall'ereditarietà

**preserveInheritance** per conservare le regole di accesso ereditate

[38] sigla da sapere per l'esame...

**PersistKeyInCsp (P)** se memorizzare la chiave nel CSP per riutilizzarla in seguito. In tal caso è necessario passare un'istanza di *CspParameters* a RSACryptoServiceProvider in cui è stata specificata la proprietà CspParameters.KeyContainerName

**Decrypt (M)** decifra dati. Prende un array di byte cifrato e restituisce l'array di byte decifrato

**Encrypt (M)** cifra dati. Prende un array di byte decifrato e restituisce l'array di byte cifrato

**ExportParameters (M)** esporta le chiavi come struttura RSAParameters. Se viene passato true vengono salvate entrambe le chiavi, se false solo la chiave pubblica

**ImportParameters (M)** importa le chiavi dalla struttura RSAParameters

**FromXmlString (M)** importa una coppia di chiavi da una stringa xml

**ToXmlString (M)** esporta una coppia di chiavi da una stringa xml. Se viene passato true vengono salvate entrambe le chiavi, se false solo la chiave pubblica

A sua volta, la struttura RSAParameters è formata da:

**D** la chiave privata

**Exponent** anche conosciuto come $e$: la parte corta della chiave pubblica

**Modulus** anche conosciuto come $n$: la parte lunga della chiave pubblica

#### 14.3.3 Integrità dei dati tramite hash

Sono supportati i seguenti algoritmi hashing nonkeyed:

**MD5** Message Digest a 128 bit

**RIPEMD160** rimpiazzo di MD5 a 160 bit

**SHA1** Secure Hash Algorithm a 160 bit

**SHA256** Secure Hash Algorithm a 256 bit

**SHA384** Secure Hash Algorithm a 384 bit

**SHA512** Secure Hash Algorithm a 512 bit

Un cracker anche se non può accedere al contenuto dei valori è comunque in grado di modificarli. Per impedire questa possibilità sono supportati anche i seguenti algoritmi basati su chiave:

**HMACSHA1** Message Authentication Code tramite SHA1. Utilizza chiavi di qualunque dimensione e produce una sequenza hash di 20 byte

**MACTripleDES** Message Authentication Code tramite TripleDES. Utilizza chiavi di lunghezza 8, 16 o 24 byte e produce una sequenza hash di lunghezza 8 byte

Per calcolare un hash bisogna passare l'array di byte a (Keyed)HashAlgorithm.ComputeHash e ottenere l'array di byte (Keyed)HashAlgorithm.Hash. Nel caso di hashing keyed è necessario calcolare la key da una password utilizzando sempre Rfc2898DeriveBytes.

#### 14.3.4 Firma Digitale

La firma digitale è il valore che si può apporre a un dato elettronico per provare che è stato creato da qualcuno in possesso di una specifica chiave privata, consentendo in tal modo di autenticare l'identità dell'utente e proteggere l'integrità dei dati, ma non la riservatezza (per questo è necessario cifrare i dati).

Sono disponibili le due classi per la crittografia asimmetrica per generare e verificare le firme digitali: DSACryptoServiceProvider e RSACryptoServiceProvider. Ognuna implementa i seguenti quattro metodi:

**SignHash** crea una firma digitale basata sull'hash

**SignData** crea una firma digitale prima creando l'hash del file e successivamente generando la firma basata sul valore hash ottenuto

**VerifyHash** verifica una firma digitale basata sull'hash di un file

**VerifyData** verifica una firma digitale dato l'intero contenuto di un file

I metodi Sign utilizzano la chiave privata dell'utente, mentre i metodi Verify utilizzano la chiave pubblica.

# 15 Interoperabilità

L'interoperabilità è il generico nome utilizzato per interagire con unmanaged code a partire da un managed code. Questo è necessario poiché non tutte le API di Windows prevedono un wrapper nel framework .NET 2.0. Inoltre moltre aziende utilizzano codice legacy che sarebbe troppo costoso riscrivere da zero. In questi casi è necessario far uso del namespace *System.Runtime.InteropServices*.

## 15.1 Da COM a .NET

Prima dell'avvento del framework .NET, COM era il principale framework per gli sviluppatori Windows per interagire con il sistema operativo Windows. Common Language Runtime espone gli oggetti COM tramite un proxy denominato *Runtime Callable Wrapper* (*RCW*). Benché l'RCW appaia ai client .NET come un normale oggetto, la sua funzione principale consiste nell'effettuare il marshalling tra un client .NET e un oggetto COM. Il wrapper standard applica le regole di marshalling incorporate. Quando ad esempio un client .NET passa come parte di un argomento un tipo String a un oggetto gestito, il wrapper converte la stringa in un tipo BSTR. Nel caso in cui l'oggetto COM dovesse restituire un valore di tipo BSTR, il relativo chiamante gestito riceverebbe una stringa. Sia il client che il server inviano e ricevono dati rispettivamente noti. Gli altri tipi non richiedono alcuna conversione. Un wrapper standard, ad esempio, trasferirà sempre un intero di 4 byte tra il codice gestito e quello non gestito senza operare alcuna conversione di tipo.

I componenti COM, a differenza di quelli .NET, devono essere registrati prima di essere usati tramite *Regsvr32.exe*. Dopo la registrazione possono essere importati o tramite Visual Studio o tramite *TlbImp.exe*. In realtà ci sono alcuni problemi di compatibilità: ad esempio in Visual Basic è possibile specificare parametri opzionali mentre in C# non è possibile. Diventa quindi necessario passare sempre qualcosa, anche se quel

qualcosa non servirà a nulla. Per ovviare a questo problema è possibile passare la proprietà statica *Type.Missing*.

Altri strumenti utilizzati sono (compresi quelli già visti):

**TlbImp.exe** consente di convertire le definizioni dei tipi presenti in una libreria dei tipi COM nelle definizioni equivalenti di un assembly di Common Language Runtime (*da COM a .NET*). L'output di Tlbimp.exe è un file binario (assembly) che contiene *metadati* per il runtime corrispondenti ai tipi definiti all'interno della libreria dei tipi originale. È possibile esaminare questo file con strumenti quali *Ildasm.exe*

**Ildasm.exe** (Intermediate Language Disassembler) mostra una rappresentazione visuale dell'Intermediate Language (IL)

**TlbExp.exe** consente di generare una *type library* che contiene le definizioni dei tipi definiti nell'assembly, utilizzabili ad esempio da parte di componenti COM (*da .NET a COM*). Le type library vengono generate *ma non registrate*. Questo comportamento è opposto a quello di (Regasm.exe), che effettua *sia la generazione che la registrazione* delle librerie dei tipi

**Regasm.exe** (Assembly Registration Tool) legge i metadati all'interno di un assembly e aggiunge al Registro di sistema le voci necessarie per consentire ai client COM di creare classi di .NET Framework in modo trasparente. Quando una classe è stata registrata, qualsiasi client COM può utilizzarla come se si trattasse di una classe COM.

Quando una classe è stata registrata, qualsiasi client COM può utilizzarla come se si trattasse di una classe COM. La classe viene registrata una sola volta, ossia al momento dell'installazione dell'assembly. Non è possibile creare da COM istanze di classi nell'assembly finché tali classi non sono state effettivamente registrate.

Quando si specifica l'opzione "`/tlb`", viene generata e *registrata* una libreria dei tipi in cui sono descritti i tipi presenti nell'assembly (*da .NET a COM*). Le librerie dei tipi generate vengono inserite nella directory di lavoro corrente o nella directory specificata per il file di output. L'utilizzo dell'opzione "/tlb" equivale all'utilizzo di entrambe le utilità Tlbexp.exe e Regasm.exe, con l'eccezione che Tlbexp.exe *non registra* la libreria dei tipi che produce.

La differenza tra Regasm e Regsvr32 è che mentre il primo crea e registra un wrapper COM da un assembly .NET, il secondo invece registra componenti COM auto-registrabili come OLE, DLL, OCX

**Regedit.exe** consente di cercare e gestire elementi del registro appartenenti a componenti COM

Alcuni linguaggi, quali ad esempio C++, consentono di generare eccezioni di qualsiasi tipo (ad esempio throw(int)). Con altri, quali C#, è necessario che ogni eccezione generata sia derivata dalla classe Exception CLS-Compliant. Per garantire la compatibilità fra i linguaggi, Common Language Runtime (CLR) esegue l'incapsulamento degli oggetti che non derivano dalla classe Exception in un oggetto RuntimeWrappedException, che a sua volta deriva da Exception e che possiede la proprietà RuntimeWrappedException.WrappedException di tipo Object che contiene l'eccezione avvolta. Questo comportamento può essere disabilitato tramite l'attributo RuntimeCompatibility (in tal caso è necessario catturare eccezioni non CLS tramite un semplice catch (...)).

Le limitazioni dell'interoperabilità con componenti COM sono i seguenti:

- nessun supporto per membri static
- solo costruttori no-args
- membri che offuscano membri nella classe base non sono riconoscibili
- problemi di portabilità a causa della dipendenza dal registro di sistema

## 15.2 Da .NET a COM

Quando un client COM unmanaged chiama un oggetto .NET, Common Language Runtime crea l'oggetto managed e un *COM Callable Wrapper* (*CCW*) per l'oggetto. Incapaci di fare direttamente riferimento a un oggetto .NET, i client COM utilizzano il CCW come un proxy per l'oggetto managed. Più client COM possono mantenere un riferimento allo stesso CCW mentre il CCW mantiene a sua volta un solo riferimento all'oggetto managed che implementa l'interfaccia ed è sottoposto alla procedura di Garbage Collection.

Per consentire questo comportamento è necessario:

- selezionare nelle opzioni di compilazione l'opzione "Register For COM Interop"
- controllare cosa rendere visibile ai componenti COM tramite l'attributo:

  ```
  [assembly: ComVisible(false)]
  ```

  in cui prende precedenza la più granulare (cioè la più annidata)
- essere conforme alle seguenti linee guida:
  - tutte le classi devono utilizzare il costruttore di default senza parametri
  - ogni tipo/membro da esporre deve essere pubblico e non statico
  - le classi astratte non devono essere esposte

Una volta rispettato questi criteri, è possibile esportare il dll dell'assembly tramite *TlbExp.exe* e registrarlo tramite *Regasm.exe*. Alle classi o agli assembly è possibile applicare l'attributo *ClassInterface*, che determina se Tlbexp.exe genera automaticamente un'interfaccia di classe per la classe con attributi. Un'*interfaccia di classe* ha lo stesso nome della classe, ma preceduto da un carattere underscore. Quando viene esposta, l'interfaccia di classe contiene tutti i membri public (non static) della classe, inclusi i membri ereditati dalla classe base. Le classi managed non possono accedere a un'interfaccia di classe e non è necessario che lo facciano, in quanto possono accedere direttamente ai membri della classe. Tlbexp.exe genera un identificatore di interfaccia univoco (IID) per l'interfaccia di classe. I tipi previsti sono i seguenti:

**AutoDispatch** la classe supporta solo l'associazione tardiva dei client COM. Un'interfaccia dispatch per la classe viene esposta automaticamente ai client COM su richiesta

**None** non viene generata alcuna interfaccia di classe. Unico modo per l'esporre la funzionalità tramite interfacce implementate in modo esplicito dalla classe (es.Comparable)

**AutoDial** viene generata automaticamente un'interfaccia di classe duale per la classe ed esposta a COM. Le informazioni sul tipo sono prodotte per l'interfaccia di classe e pubblicate nella libreria dei tipi. L'utilizzo di AutoDual è fortemente sconsigliato a causa delle limitazioni nel controllo delle versioni

### 15.3 Da Unmanaged a .NET

Quando vi è la necessità di invocare API unmanaged nel framework .NET si usa il Platform Invoke, anche detto *P/Invoke*, nel namespace System.Runtime.InteropServices nel seguente modo:

1. crea un nuovo metodo statico con la signature del metodo da invocare
2. decora il metodo con l'attributo *DllImport* specificando la libreria da invocare, ad esempio:

   ```
   [DllImport("user32.dll")]
   private static extern Int32 GetWindowText(IntPtr hWnd,
         StringBuilder textValue, Int32 counter);
   ```

3. invoca il metodo dal tuo codice

Per ovviare alla mancanza di eleganza di questi metodi è conveniente avvolgere metodi unmanaged in metodi managed in modo da semplificare l'interfaccia con sintassi più "C#-friendly".

E' possibile creare strutture dati elementari o complessi da passare a un metodo unmanaged, in modo tale che un puntatore che prevede diversi valori possa essere trasformato in un enum. Per convertire i tipi di dati semplici l'attributo *MarshalAs*, applicabile a proprietà o a parametri consente di specificare il tipo da cui dovrebbe essere convertito, ad esempio:

```
[MarshalAs(UnmanagedType.LPStr)]
public String FirstName;
```

Il marshal invece di strutture unmanaged deve seguire delle regole di convenzioni per garantire performance la cui conformità rimane responsabilità dello sviluppatore. Per controllare manualmente il marshalling di una struttura si utilizza l'attributo *StructLayoutAttribute*, il cui costruttore prende uno dei seguenti valori:

**LayoutKind.Auto** delega completamente il controllo alla CLR

**LayoutKind.Sequential** preserva il layout specificato dallo sviluppatore

**LayoutKind.Explicit** layout esplicitamente dichiarato tramite l'attributo FieldOffset

E' possibile convertire puntatori a metodi delegate in modo da gestire callback in modo type-safe. Per gestire le eccezioni è necessario ottenere l'ErrorCode tramite *Marshal.GetLastWin32Error()*.

# 16 Reflection

Uno dei benefici più grandi del CLR è la ricchezza di informazioni di tipo disponibili a runtime, interrogabili grazie al sistema reflection. Questa funzionalità consente di creare sistemi fortemente dinamici e creare architetture a plug-in.

### 16.1 Assembly

Un assembly contiene le seguenti informazioni:

- metadati dell'assembly (nome, versione, strong name, culture, ...)
- metadati dei tipi (namespace, classi, metodi, proprietà, costruttori, parametri, ...)
- codice IL (Intermediate Language)
- risorse (stringhe, immagini, file, ...)

Un assembly può essere contenuto anche in file separati, utile per consentire un download incrementale degli elementi necessari. Deve comunque esistere un assembly principale che contiene come minimo i metadati dell'assembly. I file esterni possono contenere risorse e/o moduli che sono contenitori di tipi.

La classe *Assembly* supporta una serie di metodi statici per creare un'istanza della classe, tra cui:

**GetEntryAssembly** restituisce l'assembly che contiene il metodo startup

**GetExecutingAssembly** restituisce l'assembly del codice attualmente in esecuzione

**GetCallingAssembly** restituisce l'assembly del metodo un livello sopra lo stack

**Load(From)** carica un assembly nell'AppDomain corrente (al path specifico)[39]

**ReflectionOnlyLoad(From)** carica un assembly, ma solo per interrogazioni, nell'AppDomain corrente (al path specifico)

Una volta ottenuta l'istanza Assembly sono presenti i seguenti metodi:

**CreateInstance** crea un'istanza di un tipo specifico presente nell'assembly

**GetModules** restituisce tutti i moduli. Di base sempre uno esiste.

**GetTypes** restituisce tutti i tipi presenti in tutti i moduli. Semplicemente una scorciatoia per:

```
Module[] mods = a.GetModules();
foreach (Module m in mods) {
/* memorizza m.GetTypes() */
}
```

quindi è come dire: l'assembly è un contenitore di moduli e i moduli sono i contenitori di tipi

I metadati dell'assembly sono impostabili nella forma:

[39] nota per l'esame: *AppDomain non contiene il metodo LoadFrom*

```
[assembly: AssemblyXXXXXX(valore)]
```

dove *XXXXXX* rappresenta l'informazione da specificare:

**AlgorithmId** quale algoritmo hash utilizzare nella lettura del file hash del manifest

**Company** l'azienda che ha prodotto l'assembly

**Configuration** quale configurazione utilizzare, come "DEBUG" o "RELEASE"

**Copyright** informazioni di copyright

**Culture** normalmente neutral, ma in caso di assembly satellite è necessario dichiarare la specific culture

**DefaultAlias** semplifica il nome dell'assembly

**DelaySign** dichiara che l'assembly verrà firmato dopo la compilazione e spuntato come strong assembly

**KeyFile** specifica la chiave da utilizzare per la firma (file *.snk generato da *sn.exe*[40])

**Description** descrizione per uso umano

**Flags** imposta uno o più valori AssemblyNameFlags, tra cui ottizzazione JIT, creazione chiave pubblica

**Title** titolo dell'assembly

**Trademark** marchio dell'assembly

**Version** versione dell'assembly. Composta da: "MajorVersion.MinorVersion.BuildNumber.Revision" ad esempio "1.2.3.4". Utilizzando ad esempio 1.2.*.* il BuildNumber è un numero autoincrementato che si aggiorna giornalmente, mentre la Revision è un numero generato casualmente. Non è possibile specificare Revision e lasciare l'asterisco su BuildNumber

**FileVersion** versione dell'assembly, accessibile dal filesystem

**InformationalVersion** la versione dell'assembly, utilizzata solo a scopo di informazione, ignorata per il versioning dal runtime

Per ottenere tali attributi si utilizza sempre la classe Assembly, che implementa l'interfaccia *ICustomAttributeProvider*. Tale interfaccia prevede un metodo GetCustomAttribute con un parametro booleano per indicare se ottenere anche attributi ereditati, ma chiaramente per la specifica implementazione in Assembly questo parametro è inutilizzato.

## 16.2 Type

La classe *Type* rappresenta un singolo tipo (class, enum, interface, primitivi, value type) e consente di ottenerne metodi, proprietà, eventi, interfacce e albero di ereditarietà. Tutti gli elementi di un tipo sono genericamente indicati come *Member*. Per ottenere oggetti Type sono possibili diverse strade:

- dalla classe Assembly
- dalla classe Module
- da istanze di un oggetto (Object.GetType)
- dalla keyword typeof

Il tipo è il tipo concreto, indipendentemente dal fatto che sia stato eseguito un downcast:

```
Animal cane = new Dog();
cane.GetType().Name; // Dog
```

Per ottenere tutti i membri di un tipo si usa Type.GetMembers() specificando quale tipo di elementi restituire tramite un BindingFlags. Nello specifico, se servono solo metodi si usa Type.GetMethods o Type.GetConstructors. Gli oggetti restituiti sono *MethodInfo* e *ConstructorInfo*, che ereditano da *MethodBase* e che a sua volta contiene il metodo MethodBase.GetMethodBody. L'istanza *MethodBody* ottenuta consente tramite *LocalVariables* di ottenere le variabili allocate e tramite GetILAsByteArray di ottenere un array di byte che coincide con il codice IL.

## 16.3 Codice dinamico

Il sistema reflection consente di creare oggetti dinamicamente, persino da assembly che prima non erano referenziati. Anche se è più difficile di scrivere codice verificato dal compilatore, a volte risulta necessario[41].

Per questo è necessario istanziare oggetti e invocare metodi tramite:

**GetConstructor(Types)** restituisce un ConstructorInfo corrispondente al costruttore con la signature specificata, invocabile tramite *ConstructorInfo.Invoke(object[] params)*

**GetMethod(name, Types)** restituisce il MethodInfo corrispondente, invocabile tramite *MethodInfo.Invoke(instance, object[] params)*. In caso di metodi statici il valore di instance deve essere nullo

**GetProperty(name)** restituisce il PropertyInfo corrispondente, invocabile tramite *PropertyInfo.GetValue(instance, object[] index)*

Il sistema reflection non è solo limitato a rilevare ed eseguire informazioni da assembly e tipi esistenti, ma consente perfino di creare dinamicamente codice, tipi e assembly da eseguire al volo o da memorizzare su disco. Questo è possibile tramite il namespace System.Reflection.Emit che contiene una serie di classi *XxxxxBuilder* utili per costruire assembly, tipi, metodi, ... Ogni classe Builder deriva dalla sua controparte: AssemblyBuilder estende Assembly, ConstructorBuilder estende Constructor e così via. In ordine gli step necessari sono i seguenti:

1. creare un AssemblyBuilder tramite il metodo statico AppDomain.DefineDynamicAssembly che richiede un AssemblyName per i metadati e un AssemblyBuilderAccess per il tipo di creazione (ReflectionOnly, Run, Save, RunAndSave)

[40] da non confondere con *signcode.exe*, che invece firma l'*assembly* con uno strong name (non un file)

[41] si immagini un sistema a plugin che deve avviare una classe conforme all'interfaccia *PluginStarter*: il sistema non conosce in anticipo i plugin e verranno quindi referenziati solo in seguito tramite reflection

2. creare un ModuleBuilder tramite AssemblyBuilder.DefineDynamicModule specificando nome e filename
3. creare un TypeBuilder tramite ModuleBuilder.DefineType specificando nome, tipo (class, enum, struct), visibilità (public, private), classe base e interfacce implementate.
4. creare un ConstructorBuilder tramite TypeBuilder.Define(Default)Constructor
5. ottere un ILGenerator tramite ConstructorBuilder.GetILGenerator
6. inserire linee di codice IL tramite ILGenerator.Emit specificando un OpCodes
7. memorizzare su disco l'assembly tramite AssemblyBuilder.Save specificando il percorso della dll

In modo analogo è possibile creare anche MethodBuilder, FieldBuilder, PropertyBuilder. Per dichiarare una proprietà è necessario:

1. creare un PropertyBuilder $p$ tramite TypeBuilder.DefineProperty
2. creare MethodBuilder con nome "get_NomeProperty" o "set_NomeProperty" e dichiarare modificatori SpecialName e HideBySig
3. associare i metodi get e set con il PropertyBuilder $p$ tramite $p$.SetGetMethod o $p$.SetSetMethod

# 17 Mail

Il namespace System.Net.Mail consente di creare e inviare messaggi email. Il processo di creazione di una mail è il seguente:

1. creare un oggetto MailMessage $m$. Di default $m$.Body è sempre plain text, a meno che non sia impostato $m$.IsBodyHtml a true
   - se non si è specificato i destinatari $m$.To nel costruttore di MailMessage è necessario creare oggetti MailAddress e aggiungerli a $m$. In aggiunta si possono aggiungere destinatari anche in $m$.Cc o in $m$.Bcc.
     E' possibile inserire destinatari multipli anche semplicemente specificando nel costruttore una singola stringa contenente i destinatari separati da virgola o punto e virgola
   - se è necessario fornire viste multiple (plain e HTML), creare oggetti AlternateView e aggiungerli a $m$.AlternateViews.
     In caso di messaggi HTML è possibile fondere immagini all'html creando una LinkedResource $l$ aggiungendola ad AlternateView.LinkedResources. La proprietà $l$.ContentId identifica la risorsa nell'ambito dell'html ed è referenziabile tramite la sintassi:
     ```
     <img src="cid:CONTENTID">
     ```
   - se è necessario allegare file, creare uno o più oggetti Attachment e aggiungerli a $m$.Attachments
2. creare un oggetto SmtpClient $s$ specificando il server SMTP
   - se il server SMTP richiede autenticazione, aggiungere le Credential a $s$, tramite il metodo statico CredentialCache.DefaultNetworkCredentials o creando appositamente una NetworkCredential
   - è possibile abilitare la cifratura SSL impostando true a $s$.EnableSsl
3. passa $m$ a $s$.Send. Se si vuole inviare in modo asincrono utilizzare $s$.SendAsync creando un metodo per rispondere all'evento $s$.SendCompleted, in cui è possibile cancellare l'operazione tramite $s$.SendAsyncCancel

Nell'invio di una mail possono verificarsi le seguenti eccezioni:

**InvalidOperationException** nessun server definito

**SmtpException** impossibile trovare il server, utente non valido o altri problemi di comunicazione

**SmtpFailedRecipientException** si sta inviando il messaggio a un destinatario localizzato internamente al mail server specificato ma tale destinatario non esiste. Questo tipo di errore è quindi direttamente segnalabile dal server SMTP ma è riportato solo su richiesta sincrona di Send

# 18 Globalizzazione

Gli strumenti di globalizzazione nel namespace System.Globalization consentono di creare applicazioni che funzionano in modo equivalente in Giappone e in Gran Bretagna senza cambiamenti nel codice.

Per ottenere informazioni sul contesto culturale dell'applicazione in esecuzione si utilizza la classe CultureInfo, utile per:

- eseguire confronti lessicografici, numerici e basati su date
- formattare opportunamente le date
- ottenere e utilizzare risorse

Una cultura può essere:

**Invariant Culture** culture-insensitive. Anche se si basa sull'inglese non deve essere confuso come "lingua inglese" poiché il suo utilizzo al posto della cultura en-EN può dar luogo a errori. Utile per eseguire confronti indipendenti dal contesto culturale: è possibile ad esempio confrontare la data odierna indipendentemente dal formato utilizzato[42].

**Neutral Culture** English (en), French (fr), Spanish (sp), Italian (it) sono culture neutrali, che cioè non hanno ancora alcuna relazione con paesi o regioni. Utile a distinguere differenze regionali: l'inglese americano è differente da quello britannico. L'utilizzo della cultura neutrale dovrebbe essere evitato per le stesse ragioni dell'invariant

[42] ad esempio, in Italia si usa la convenzione GG/MM/HHHH mentre in Inghilterra si usa MM/GG/HHHH

**Specific Culture** la più precisa categoria ed è rappresentata da una cultura neutrale, un segno meno e l'abbreviazione della cultura specifica, come ad esempio "en-US". Dovrebbe essere utilizzata solo questa. Le uniche due eccezioni sono "zh-CHS" (Simplified Chinese) e "zh-CHT" (Traditional Chinese) che sono invece Neutral Culture.

Per ottenere la cultura corrente *CultureInfo* si utilizza la proprietà Thread.CurrentThread.Current(UI)Culture che restituiscono la cultura corrente (UI: utilizzata solo dalla UI, non dal sistema e manipolabile solo allo startup). Non si deve cablare informazioni culturali in un testo:

```
string money = "$100,000.00"; // NO
string money = (100000).ToString("C"); // OK
```

dove la "C" sta per "Currency".

Si possono ottenere maggiori informazioni sulla CultureInfo *c* tramite *RegionInfo*, creabile passando l'identificativo numerico *c*.LCID o l'identificativo testuale *c*.Name. Grazie ad esso, la proprietà RegionInfo.Name restituisce le due lettere ISO 3166 corrispondenti al country/region code.

Per risolvere confronti numerici o di date si utilizza DateTimeFormatInfo e NumberFormatInfo, ottenibili tramite CultureInfo. CultureInfo.CompareInfo consente di confrontare due stringhe tenendo presente le differenze culturali tramite metodi Compare o IndexOf. Ad esempio, in tedesco "Strass" e "Straß" vengono considerati uguali. Si può raffinare il confronto passando al metodo Compare anche un CompareOptions specificando ad esempio l'opzione IgnoreCase. E' possibile convertire una forma a stringa in un DateTime equivalente tramite il metodo *DateTime.(Try)Parse(Exact)* che (prova) a convertire la stringa (con un pattern specifico).

Per creare le proprie culture personalizzate si usa *CultureAndRegionInfoBuilder*, ereditando opzionalmente oggetti preesistenti RegionInfo e/o CultureInfo e definendo un valore nell'enum CultureAndRegionModifiers:

**Neutral** neutral culture

**None** specific culture

**Replacement** rimpiazzamento di una culture preesistente o di un Windows Locale

La formattazione dei numeri può essere manipolata tramite la proprietà CultureAndRegionModifiers.NumberFormat. Nel dettaglio, il procedimento richiesto per creare e usare la cultura personalizzata è il seguente:

1. usare un oggetto CultureAndRegionInfoBuilder per definire e nominare una cultura personalizzata
2. modificare le proprietà dell'oggetto CultureAndRegionInfoBuilder per soddisfare i requisiti culturali richiesti
3. invocare il metodo Register per registrare la cultura
4. creare un oggetto CultureInfo specificando il nome della cultura personalizzata

# 19 Varie ed eventuali...

Seguono alcune informazioni necessarie per i test:

- dato un Thread *t*, il metodo *t*.StartLowPriority() avvia il thread con bassa priorità ed è analogo a impostare *t*.Priority a ThreadPriority.Lowest
- Double possiede il seguente metodo *Double.ToString(IFormatProvider)*, ad esempio:

  ```
  double d = 65533.22;
  CultureInfo ci = new CultureInfo("da-DK");
  d.ToString(ci);
  ```

- DateTime possiede il metodo *DateTime.ToString(String format, IFormatProvider)*
- la classe StringBuilder (stranamente) non implementa l'interfaccia Comparable
- gli archivi isolated storage possono essere mostrati ed eliminati con *storeadm.exe*
- l'username è ottenibile con *System.Environment.GetEnvironmentVariable("username")* e con *System.Environment.UserName*
- il metodo IAsyncResult.AsyncWaitHandle.WaitOne consente di bloccare il thread fino alla fine del metodo BeginXXX ed EndXXX (coincide con il metodo EndXXX)
- StreamReader consente di interagire *direttamente* con il filesystem (anche se in teoria decora uno stream)
- *PrintingPermission* non richiede la porta LPT1
- *SiteIdentityPermission* configura accessi per chi richiede uno specifico sito web
- *PublisherIdentityPermission* configura permessi basati sull'identità di uno specifico publisher software
- *ZoneIdentityPermission* configura permessi basati sul tipo di zona (internet, intranet, ...)
- *UrlIdentityPermission* configura accessi per un URL
- *IMembershipCondition* consente di ridefinire la logica per determinare se un assembly appartiene a un code group
  - *ApplicationDirectoryMembershipCondition* verifica la directory dell'applicazione
  - *SiteMembershipCondition* verifica il sito da cui ha origine
  - *PublisherMembershipCondition* verifica il certificato Authenticode X.509v3 del publisher del software. La verifica vera e propria avviene con il metodo Check
  - *GacMembershipCondition* verifica l'appartenenza al GAC
  - *UrlMembershipCondition* verifica il suo URL. Anche se simile ad ApplicationDirectoryMembershipCondition, dovrebbe essere utilizzato espressamente per assembly ottenuti tramite HTTP
- SortedList consente di accedere agli elementi sia tramite chiave poiché è un Dictionary, sia tramite indice poiché le proprietà Keys e i Values sono indicizzate

- un Attachment viene creato sia tramite una stringa contenente il percorso del file sia tramite uno Stream di lettura contenente i dati. Poiché FileStream e MemoryStream derivano da Stream *sono* utilizzati entrambi senza distinzioni, ma stranamente si dice che l'Attachment vuole dei dati che siano salvati su disco quindi per le domande MemoryStream non è accettabile, quando non è così...

- in graphics non esiste un metodo FillCircle, poiché un cerchio è un caso particolare di ellissi. Utilizzare quindi FillEllipse

- *SslProtocols.Tls* è il protocollo di connessione più sicuro supportato da .NET 2.0

- in una comunicazione è possibile determinare se entrambe le parti forniscono un valido certificato di autenticazione prima di avviare tale comunicazione utilizzando il metodo *SslStream.IsMutuallyAuthenticated*

- il metodo *ProtectedData.Protect* consente di restituire una copia protetta dell'array di byte che costituisce i dati da proteggere, mentre il metodo *ProtectedMemory.Protect* protegge internamente l'originale array di byte passato per parametro

- in caspol.exe, l'argomento "-customall PATH" indica che tutte le opzioni specificate *dopo* questa opzione verranno applicate ai criteri definiti a livello di computer e di azienda, nonché ai criteri personalizzati specificati a livello di utente specificati nel PATH. L'argomento "-listgroups" visualizza i codegroup del livello di criteri specificato *prima* o di tutti i livelli di criteri. L'argomento "-resolvegroup" mostra i gruppi di codice cui appartiene un assembly specifico

- *ArgumentException* è la classe base per tutti gli errori su argomenti di un metodo:

  - *ArgumentNullException*: il valore dell'argomento è obbligatorio
  - *ArgumentOutOfRangeException*: valore invalido dell'argomento

- *InvalidOperationException* si lancia se un'impostazione di una proprietà o un'invocazione di un metodo non è appropriato per lo stato corrente dell'oggetto