

# Laboratorio di Ingegneria del Software

## L-A

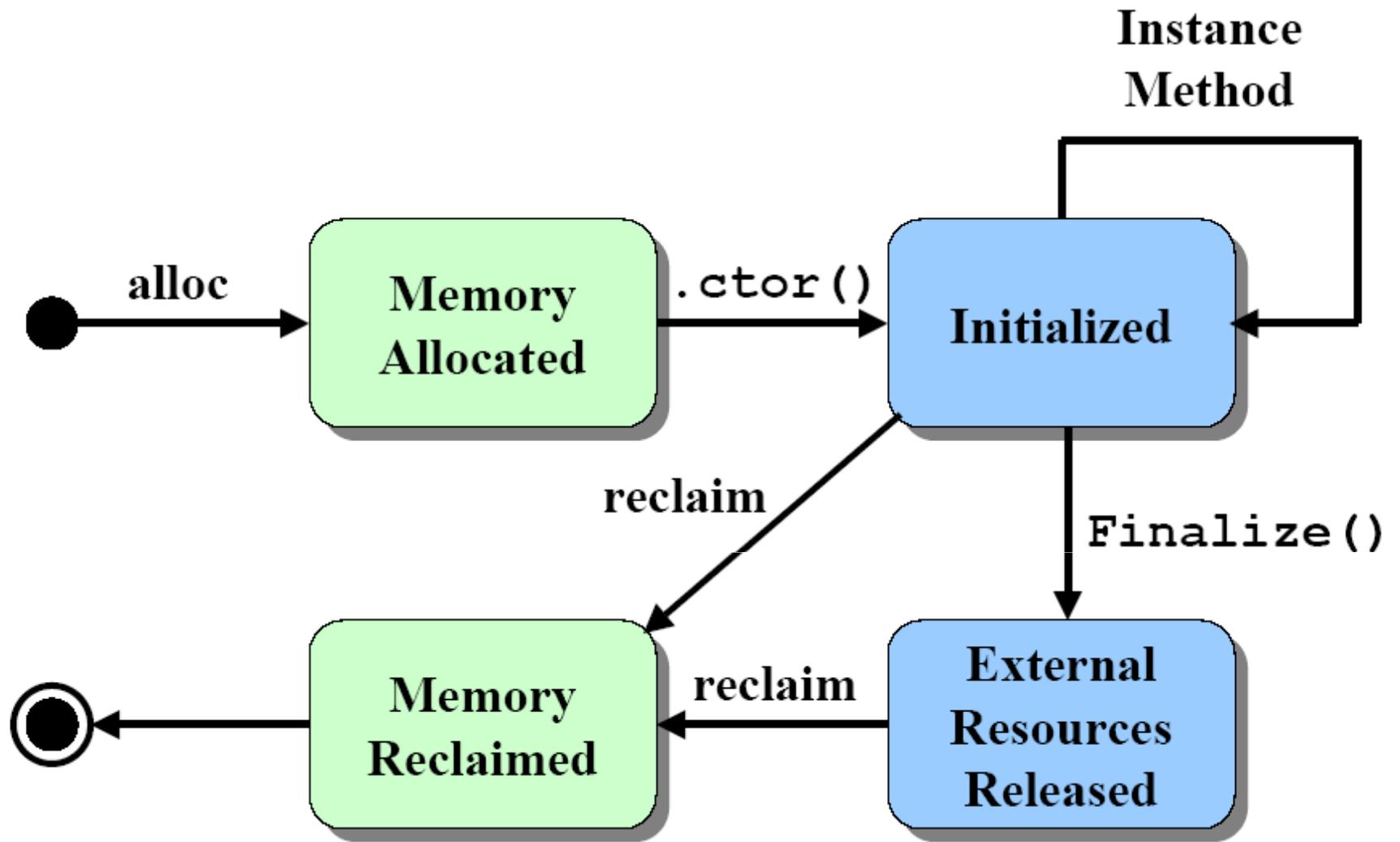
Garbage Collector



# Utilizzo di un oggetto

- In un ambiente *object-oriented*, ogni oggetto che deve essere utilizzato dal programma
  - È descritto da un tipo
  - Ha bisogno di un'area di memoria dove memorizzare il suo stato
- Passi per utilizzare un oggetto di tipo riferimento:
  - **Allocare memoria** per l'oggetto in seguito a una **new** (istruzione **newobj** di IL)
  - **Inizializzare la memoria** per rendere utilizzabile l'oggetto valori di default + costruttore
  - **Usare l'oggetto**
  - **Eeguire un clean up** dello stato dell'oggetto, se necessario **Finalize** (distruttore in C# e C++), **Dispose**, **Close**
  - **Liberare la memoria** responsabilità del **Garbage Collector (GC)**

# Ciclo di vita di un oggetto



# Allocazione della memoria

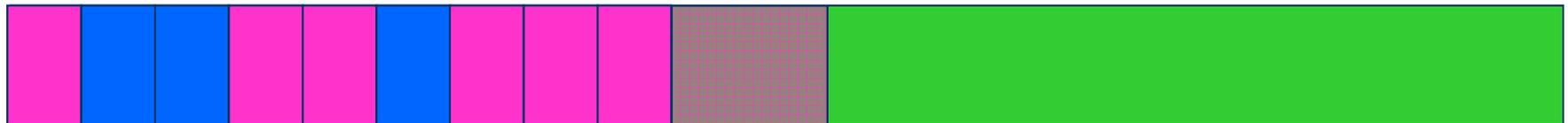
- In fase di inizializzazione di un processo, il CLR
  - Riserva una regione contigua di spazio di indirizzamento *managed heap*
  - Memorizza in un puntatore (`NextObjPtr`) l'indirizzo di partenza della regione



# Allocazione della memoria

- Quando deve eseguire una `newobj`, il CLR
  - Calcola la dimensione in *byte* dell'oggetto e aggiunge all'oggetto due campi di 32 (o 64) bit
    - Un puntatore alla tabella dei metodi
    - Un campo `SyncBlockIndex`
  - Controlla che ci sia spazio sufficiente a partire da `NextObjPtr` – in caso di spazio insufficiente:
    - *garbage collection*
    - `OutOfMemoryException`
  - `thisObjPtr = NextObjPtr;`
  - `NextObjPtr += sizeof(oggetto);`
  - Invoca il costruttore dell'oggetto (`this ≡ thisObjPtr`)
  - Restituisce il riferimento all'oggetto

# Allocazione della memoria



NextObjPtr

Tecnica di allocazione  
completamente diversa  
da quella del C/C++



Oggetti “vivi”



Oggetti non raggiungibili



Spazio libero

# Garbage Collector

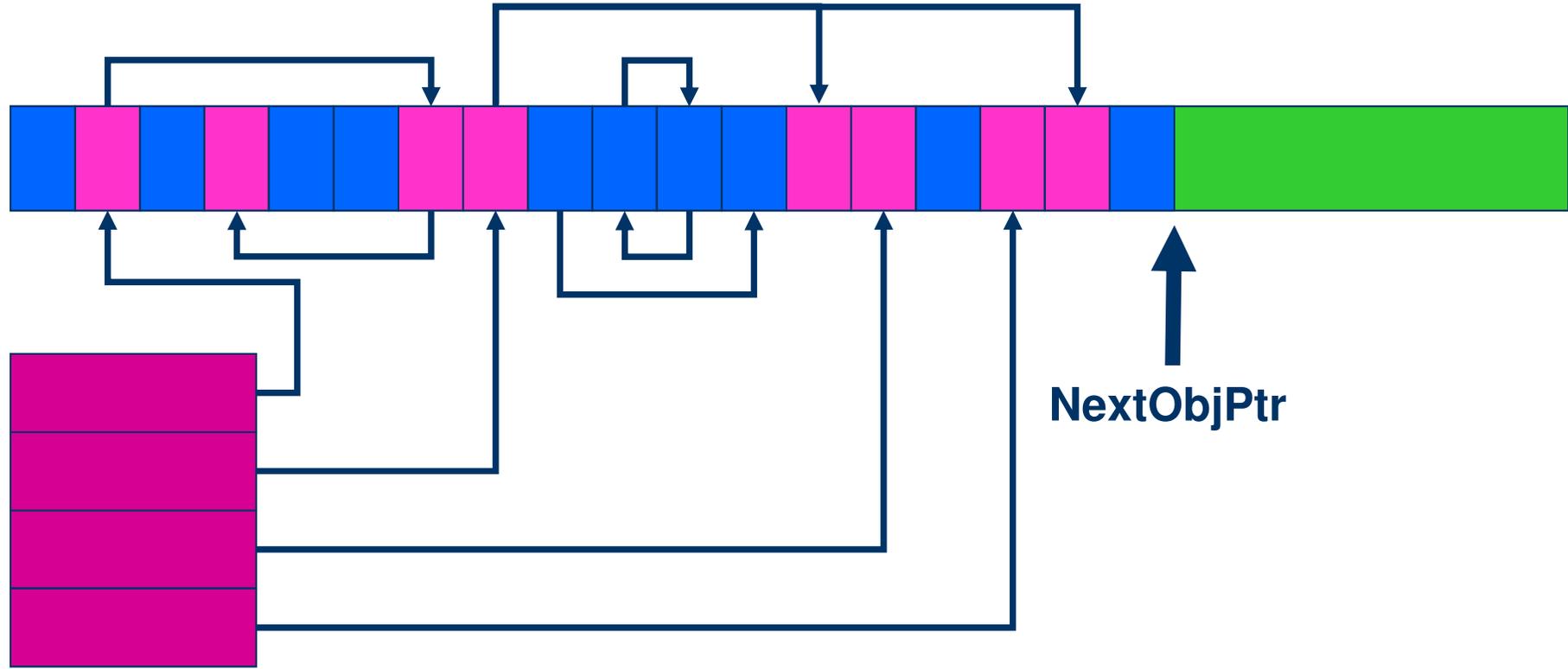
- Verifica se nell'*heap* esistono oggetti non più utilizzati dall'applicazione
  - Ogni applicazione ha un insieme di radici (*root*)
  - Ogni radice è un puntatore che contiene l'indirizzo di un oggetto di tipo riferimento oppure vale `null`
  - Le radici sono:
    - Variabili globali e *field* statici di tipo riferimento
    - Variabili locali o argomenti attuali di tipo riferimento sugli *stack* dei vari *thread*
    - Registri della CPU che contengono l'indirizzo di un oggetto di tipo riferimento
  - **Gli oggetti “vivi”** sono quelli **raggiungibili** direttamente o indirettamente dalle radici
  - **Gli oggetti *garbage*** sono quelli **NON raggiungibili** direttamente o indirettamente dalle radici

# Garbage Collector

- Quando parte, il GC ipotizza che tutti gli oggetti siano *garbage*
- Quindi, scandisce le radici e per ogni radice **marca**
  - L'eventuale oggetto referenziato e
  - Tutti gli oggetti a loro volta raggiungibili a partire da tale oggetto
- Se durante la scansione incontra un oggetto già marcato in precedenza, lo salta
  - Sia per motivi di prestazioni
  - Sia per gestire correttamente riferimenti ciclici
- Una volta terminata la scansione delle radici, tutti gli oggetti NON marcati sono non raggiungibili e quindi *garbage*

# Garbage Collector

## Fase 1: Mark



Root set

NextObjPtr

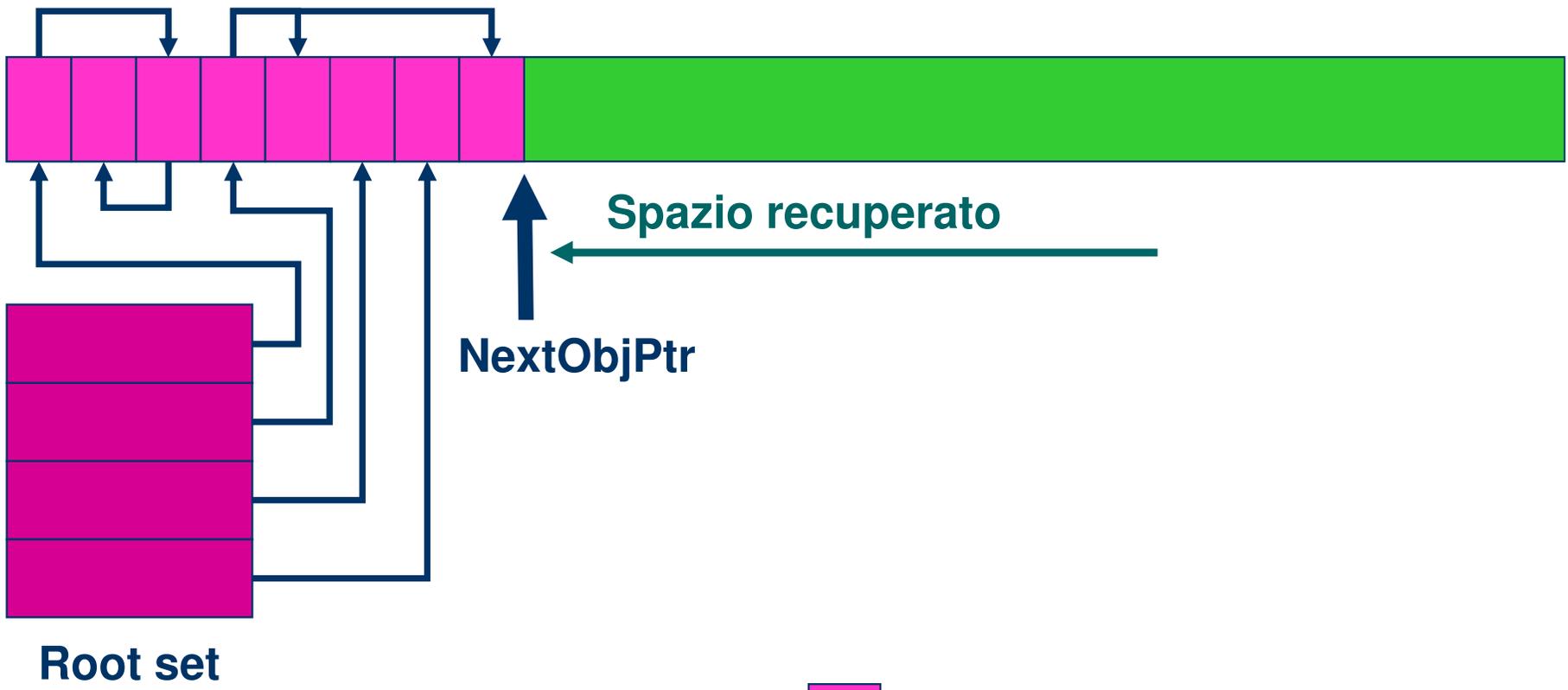
-  Oggetti "vivi"
-  Oggetti non raggiungibili
-  Spazio libero

# Garbage Collector

- Rilascia la memoria usata dagli oggetti non raggiungibili
- **Compatta** la memoria ancora in uso, **modificando nello stesso tempo tutti i riferimenti agli oggetti spostati!**
- Unifica la memoria disponibile, aggiornando il valore di **NextObjPtr**
- Tutte le operazioni che il GC effettua sono possibili in quanto
  - Il tipo di un oggetto è sempre noto
  - È possibile utilizzare i metadati per determinare quali *field* dell'oggetto fanno riferimento ad altri oggetti

# Garbage Collector

## Fase 2: Compact



-  Oggetti "vivi"
-  Oggetti non raggiungibili
-  Spazio libero

# Finalization

- Non è responsabilità del GC, ma del programmatore
- Se un oggetto contiene esclusivamente
  - tipi valore e/o
  - riferimenti a oggetti *managed*(maggior parte dei casi), non è necessario eseguire alcun codice particolare
- Se un oggetto contiene almeno un riferimento a un oggetto *unmanaged* (in genere, una risorsa del S.O.)
  - file, connessione a database, socket, mutex, bitmap, ...**è necessario eseguire del codice per rilasciare la risorsa, prima della deallocazione dell'oggetto**

# Finalization

- Ad esempio, un oggetto di tipo **System.IO.FileStream**
  - Prima deve aprire un file e memorizzare in un suo *field* l'*handle* del file (una risorsa di S.O. *unmanaged*)
  - Quindi usa tale *handle* nei metodi **Read** e **Write**
  - Infine, deve rilasciare l'*handle* nel metodo **Finalize**
- In C#
  - NON è possibile definire il metodo **Finalize**
  - È necessario definire un **distruzione** (sintassi C++)

# Finalization

```
public class OSHandle
{
    // Field contenente l'handle della risorsa unmanaged
    private readonly IntPtr _handle;
    // Proprietà che restituisce il valore dell'handle
    public IntPtr Handle
    { get { return _handle; } }

    public OSHandle(IntPtr handle)
    { _handle = handle; }
    ~OSHandle()
    { CloseHandle(_handle); }

    [System.Runtime.InteropServices.DllImport("Kernel32")]
    private extern static bool CloseHandle(IntPtr handle);
}
```

# Finalization

- Il compilatore C# trasforma il codice del distruttore

```
~OSHandle ()  
{ CloseHandle (_handle); }
```

nel seguente codice (ovviamente in IL):

```
protected override void Finalize()  
{  
    try  
    { CloseHandle (_handle); }  
    finally  
    { base.Finalize(); }  
}
```

# Finalization

- Il metodo **Finalize** dovrebbe essere utilizzato **solo per rilasciare risorse *unmanaged*** che appartengono all'oggetto su cui eseguire il metodo
- Nel metodo **Finalize** si dovrebbe **evitare di accedere ad altri oggetti *managed***
  - Potrei cercare di accedere a un oggetto sul quale il GC ha già invocato il corrispondente metodo **Finalize** e che quindi potrebbe essere in uno stato non ben definito
- Tutti i metodi **Finalize** vengono eseguiti in un *thread* dedicato (diverso da quello del GC) – pertanto, nel codice non si devono fare ipotesi sul *thread* in esecuzione

# Il pattern Dispose

- L'invocazione del metodo **Finalize** non avviene in modo deterministico
- Inoltre, non essendo un metodo pubblico, il metodo **Finalize** non può essere invocato direttamente
- Nel caso di utilizzo di risorse che devono essere rilasciate appena termina il loro uso, questa **situazione è problematica**
- Si pensi a file aperti o a connessioni a database che vengono chiusi solo quando il GC invoca il corrispondente metodo **Finalize**
- In questi casi, è di fondamentale importanza rilasciare (**Dispose**) o chiudere (**Close**) la risorsa in **modo deterministico**

# Il pattern Dispose

- I tipi che vogliono offrire questa funzionalità devono implementare il *pattern Dispose*
- Il *pattern Dispose* fissa le convenzioni da seguire quando si vuole definire un tipo in grado di offrire un servizio di ***clean up esplicito*** ai suoi utilizzatori
- Innanzi tutto, il tipo deve implementare l'interfaccia **IDisposable**

```
public interface IDisposable
{
    void Dispose();
}
```

# Il pattern Dispose

```
public class MyClass : IDisposable
{
    // Riferimenti a risorse unmanaged
    ...
    // Riferimenti a risorse managed
    // che contengono riferimenti a risorse unmanaged
    // e che quindi implementano IDisposable
    ...
    // Costruttore/i
    ...
    ~MyClass ()
    {
        Dispose (false) ;
    }
}
```

Invocazione NON  
deterministica

# Il pattern Dispose

```
// Metodo da invocare per un rilascio deterministico
public void Dispose()
{
    Dispose(true);
    // Take yourself off of the Finalization queue to prevent
    // finalization code for this object from executing a second time
    GC.SuppressFinalize(this);
}
// Metodo alternativo (opzionale) per una chiusura deterministica
public void Close()
{
    Dispose();
}
```



Invocazione deterministica

# II pattern Dispose

```
// Track whether Dispose has been called
private bool _disposed = false;

protected virtual void Dispose(bool disposing)
{
    // Synchronize threads calling Dispose/Close simultaneously
    lock (this)
    {
        if (!_disposed) // Check to see if Dispose has already been called
        {
            if (disposing)
            {
                // Dispose managed resources
            }
            // Dispose unmanaged resources
            _disposed = true;
        }
    }
}
```

# II pattern Dispose

- **Dispose** (`bool disposing`) executes in two distinct scenarios:
  - if `disposing == true`, the method has been called directly or indirectly by a user's code  
Managed and unmanaged resources can be disposed
  - if `disposing == false`, the method has been called by the runtime from inside the finalizer and **you should not reference other objects**  
Only unmanaged resources can be disposed

```
if (disposing)
{
    // Dispose managed resources
}
// Dispose unmanaged resources
```

# Il pattern Dispose

```
// Allow your Dispose method to be called multiple times,  
// but throw an exception if the object has been disposed  
// Whenever you do something with this class,  
// check to see if it has been disposed
```

```
public void DoSomething()  
{  
    if(_disposed)  
        throw new ObjectDisposedException();  
    ...  
}
```

# Il pattern Dispose in una classe derivata

```
private bool _disposed = false;
protected override void Dispose(bool disposing)
{
    lock (this)
    {
        if (!_disposed)
        {
            try
            {
                if (disposing)
                { // Dispose managed resources }
                // Dispose unmanaged resources
                _disposed = true;
            }
            finally
            { base.Dispose(disposing); }
        }
    }
}
```

# Il pattern Dispose

## dal lato del cliente – senza using

```
...  
Byte[] bytesToWrite = new Byte[] {1,2,3,4,5};  
FileStream fs = null;  
try  
{  
    fs = new FileStream("Temp.dat", FileMode.Create);  
    fs.Write(bytesToWrite, 0, bytesToWrite.Length);  
}  
finally  
{  
    if(fs != null) fs.Close();  
}  
...
```

# Il pattern **Dispose** dal lato del cliente – con **using**

```
...  
Byte[] bytesToWrite = new Byte[] {1,2,3,4,5};  
using (FileStream fs =  
    new FileStream("Temp.dat", FileMode.Create))  
{  
    fs.Write(bytesToWrite, 0, bytesToWrite.Length);  
}  
...
```

- All'uscita del blocco **using**, viene sempre invocato automaticamente il metodo **Dispose**
- Il tipo della variabile definita nella parte iniziale di **using** deve implementare l'interfaccia **IDisposable**

# Class, Component e Control

- If your class uses unmanaged resources but will not be used on a **design surface**, implement `IDisposable` or derive from a class that directly or indirectly implements `IDisposable`
- If you want a class that is **designable** (used on a design surface), you can derive from `System.ComponentModel.Component`, the base implementation of an `IComponent` type (note that `IComponent` extends `IDisposable`)
- If you want a **designable class that provides a user interface**, your class is a **control** - A control must derive directly or indirectly from one of the base control classes
  - `System.Windows.Forms.Control`
  - `System.Web.UI.Control`
- If your class is neither designable nor holds unmanaged resources, you do not need an `IComponent` or an `IDisposable` type

# Component

- A .NET Framework **Component** provides features such as
  - **Control over unmanaged resources**

The **IComponent** interface extends **IDisposable**. In its implementation of the **Dispose** method, a component must release unmanaged resources explicitly. Developers must propagate **Dispose** throughout a containment hierarchy to ensure that children of a component also free resources. Additionally, a derived component must invoke the **Dispose** method of its base class.
  - **Design-time support**

# Container

- A container logically contains one or more components that are called the container's child components
- A container is a class that implements the **System.ComponentModel.IContainer** interface or derives from a class that implements this interface
- If you are developing components and controls, you do not have to implement containers
- The designers for Windows Forms and for Web Forms are containers for Windows Forms and for ASP.NET server controls
- Containers provide services to the components and controls sited within them - at design time, controls are sited in the designer and obtain services from the designer

# Il pattern Dispose in una Form

```
public class Form1 : System.Windows.Forms.Form
{
    private System.ComponentModel.IContainer components =
        new System.ComponentModel.Container();
    ...
    protected override void Dispose(bool disposing)
    {
        if (disposing)
        {
            if (components != null)
            {
                components.Dispose();
            }
        }
        base.Dispose(disposing);
    }
}
```