

# C# e OOP

**Renata Carriero**

Renata.Carriero@icubed.it



# Principi di OOP

- Ereditarietà
- Polimorfismo
- Incapsulamento
- Astrazione

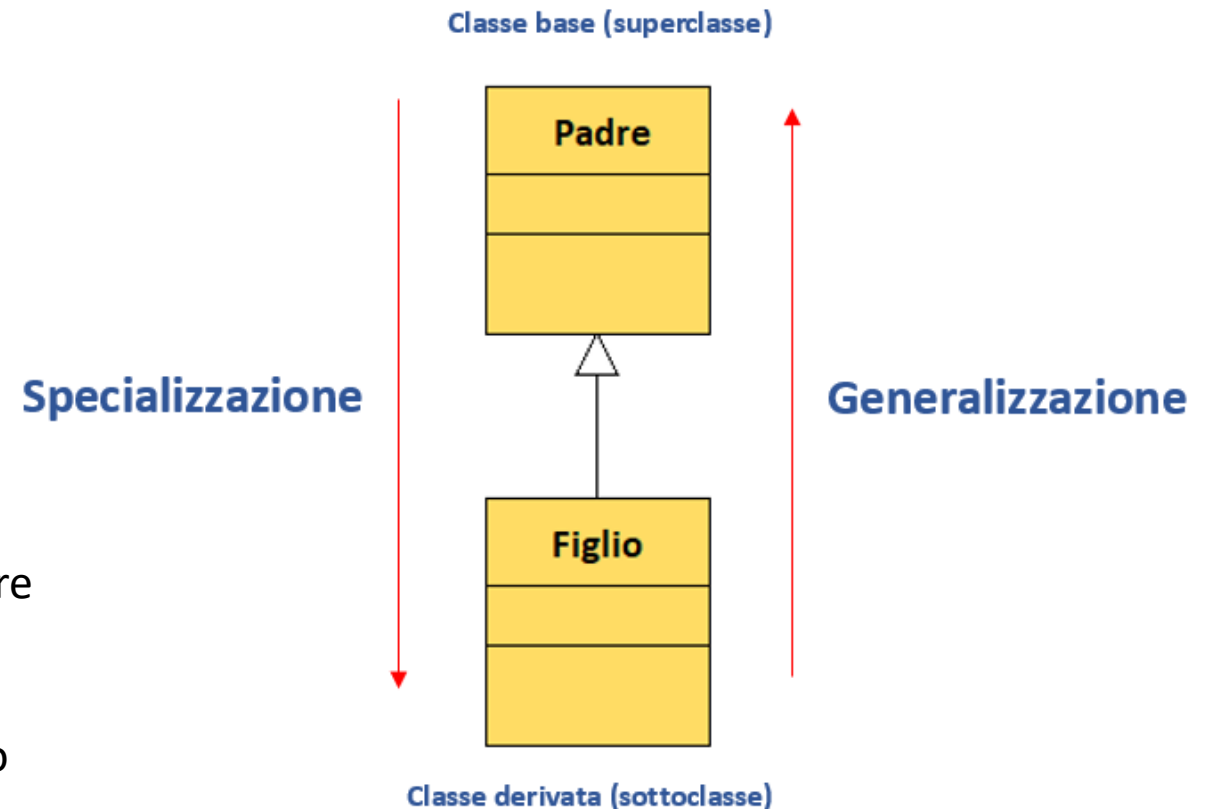
# Ereditarietà

L'ereditarietà può essere usata come meccanismo per ottenere l'estensibilità e il riuso del codice, e risulta particolarmente vantaggiosa quando si devono definire sottotipi sfruttando le relazioni is-a (è-un) esistenti nella realtà del sistema che si deve modellare.

Per esempio, partendo da una classe Persona, potremmo definire un sottotipo Studente, che è una Persona, e quindi ne condivide sicuramente alcune caratteristiche e comportamenti, ma ne aggiungerà di nuove e/o ne modificherà alcune esistenti, per specializzarle al nuovo sottotipo.

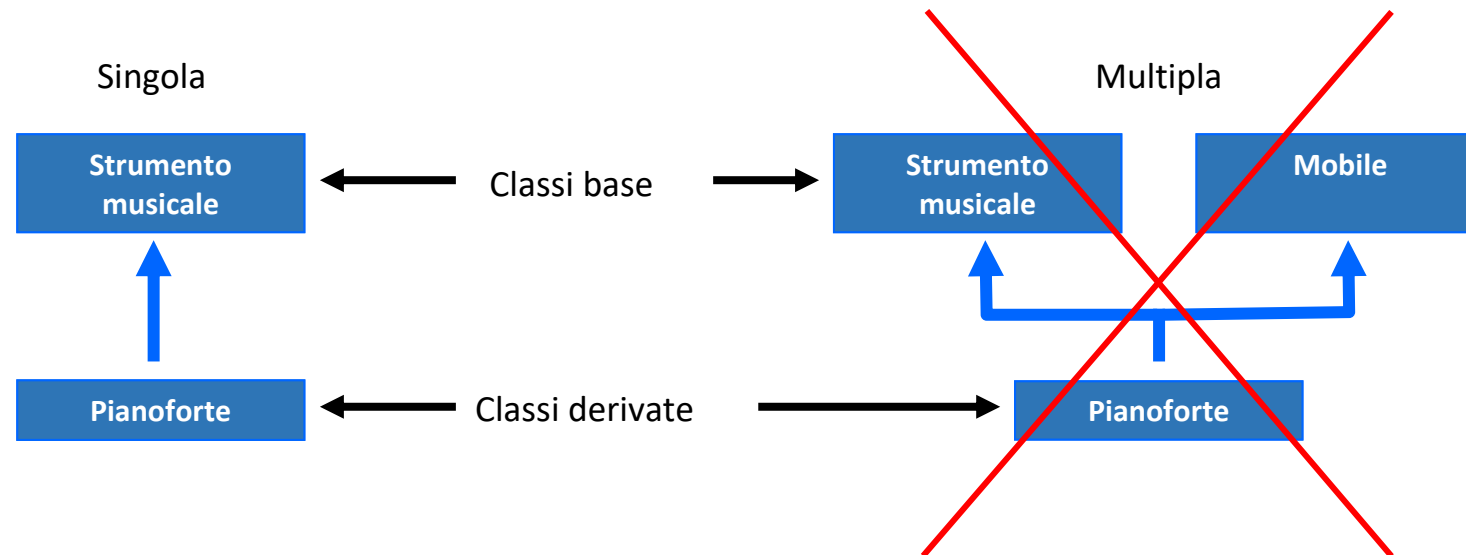
La relazione può essere letta come:

- **“è un tipo di”** (verso di generalizzazione), ed è sempre verificato (uno Studente è sempre una Persona)
- **“può essere un”** (verso di specializzazione), e non è detto che lo sia (una Persona non è detto che sia uno Studente)



# Ereditarietà singola e multipla

- Non è ammessa l'ereditarietà multipla.
- Una classe può derivare unicamente da una sola altra classe.



# Ereditarietà

- Si chiama *ereditarietà* perché la classe che deriva (**classe derivata**) può usare tutti i membri della classe ereditata (**classe base** – keyword **base**) come se fossero propri, ad eccezione di quelli dichiarati privati.

Le classi specializzate/derivate incorporano (ereditano) la struttura ed il comportamento delle classi più generali

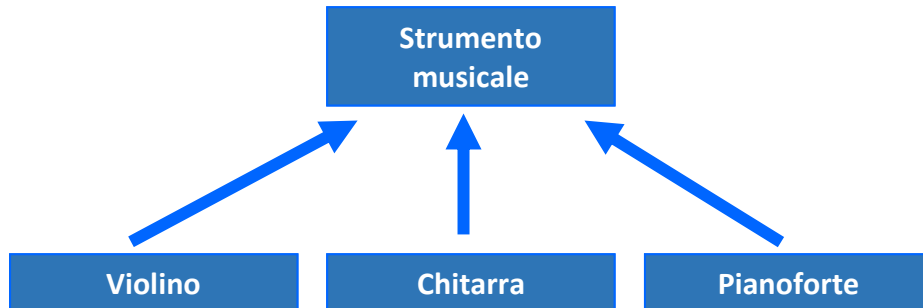
La classe derivata può:

- contenere nuovi attributi e operazioni non inclusi nella classe base
- ridefinire i metodi delle operazioni della classe base (overriding/polimorfismo).

```
public class Person {  
    protected string name;  
}  
  
public class Customer : Person {  
  
    public void ChangeName(string newName) {  
        base.name = newName;  
    }  
}
```

# Polimorfismo

- Il *polimorfismo* è la possibilità di trattare un'istanza di un tipo come se fosse un'istanza di un altro tipo.
- Il polimorfismo è subordinato all'esistenza di una relazione di derivazione tra i due tipi.
- Affinchè un metodo possa essere polimorfico, deve essere marcato come **virtual** o **abstract**.



```
public class Strumento
{
    public virtual void Accorda() { }
}

public class Violino : Strumento
{
    public override void Accorda()
    {
        base.Accorda();
    }
}

public class Orchestra
{
    public Strumento violino, chitarra, pianoforte;

    public Orchestra()
    {
        violino = new Violino();
        violino.Accorda();
    }
}
```

# La classe Object

Tutto in .NET deriva dalla **classe Object**

- Se non specifichiamo una classe da cui ereditare, il compilatore assume automaticamente che stiamo ereditando da Object

## **System.Object**

- Tutto ciò che deriva da Object **ne eredita anche i metodi**
- Questi metodi sono disponibili **per tutte le classi** che definiamo

# La classe Object

- **ToString:** converte l'oggetto in una stringa
- **GetHashCode:** ottiene il codice hash dell'oggetto
- **Equals:** permette di effettuare la comparazione tra oggetti
- **Finalize:** chiamato in fase di cancellazione da parte del garbage collector
- **GetType:** ottiene il tipo dell'oggetto
- **MemberwiseClone:** effettua la copia dell'oggetto e ritorna una reference alla copia



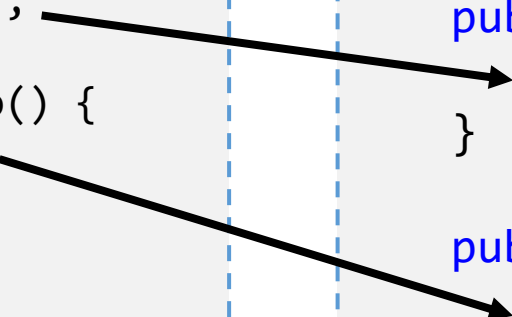
# Classi astratte e metodi virtuali

- Una classe è astratta (**abstract**) se contiene almeno un metodo astratto.
- Un metodo è astratto se dichiara la sua firma, ma non fornisce alcuna implementazione.
- Se una classe derivata deve poter provvedere una nuova implementazione di un metodo, nella classe base questo deve essere contrassegnato come virtuale (**virtual**).
- La classe derivata che voglia (o debba) fornire una implementazione sostitutiva di una classe base deve marcare con **override** il metodo.

# Classi astratte e metodi virtuali

```
public abstract class Strumento  
  
    public abstract void Accorda();  
  
    public virtual void PausaTempo() {  
        // ...  
    }  
  
}
```

```
public class Piano : Strumento  
  
    public override void Accorda() {  
        // ...  
    }  
  
    public override void PausaTempo() {  
        // ...  
    }  
  
}
```

Two black arrows originate from the left box. The first arrow starts at the 'Accorda()' line in the 'Strumento' class and points to the 'Accorda()' line in the 'Piano' class. The second arrow starts at the 'PausaTempo()' line in the 'Strumento' class and points to the 'PausaTempo()' line in the 'Piano' class.

# Modifier predefiniti

## Per tipi

- **abstract**
- **sealed**

Il tipo deve essere derivato.

Il tipo non può essere derivato.

## Per membri

- **static**

Non è un membro dell'istanza, ma del tipo.

## Per metodi

- **static**
- **virtual**
- **new**
- **override**
- **abstract**

Il metodo è associato al tipo non all'istanza.

Il tipo derivato può eseguire l'override.

Maschera il metodo del tipo base.

Ridefinisce il metodo del tipo base.

Il tipo derivato deve eseguire l'override.

# Incapsulamento

L'oggetto può essere visto come un **contenitore** di strutture dati e di procedure che le utilizzano. Gli oggetti realizzano il concetto di Incapsulamento.

Il termine **Incapsulamento** indica la proprietà degli oggetti di incorporare al loro interno sia gli attributi che i metodi, cioè le caratteristiche e i comportamenti dell'oggetto.

Si crea come una **capsula**, un contenitore concettuale, che isola l'oggetto dalle cose esterne. Si dice che gli attributi e i metodi sono incapsulati nell'oggetto, e quindi tutte le informazioni utili che riguardano un oggetto sono ben localizzate. Questa metodologia di raccogliere tutto quello che riguarda una singola entità all'interno di un oggetto è uno dei principali vantaggi che viene offerto dalla programmazione orientata agli oggetti

L'incapsulamento è proprio legato al concetto di "impacchettare" in un oggetto i dati e le azioni che sono riconducibili ad un singolo componente.

# Information hiding

Un concetto simile all'incapsulamento è l'occultamento dell'informazione, meglio noto con il termine di **information hiding**.

Tale concetto esprime l'abilità di **nascondere** al mondo esterno tutti i dettagli implementativi più o meno complessi che si svolgono all'interno di un oggetto. Il mondo esterno, per un oggetto, è rappresentato da qualunque cosa si trovi all'esterno dell'oggetto stesso.

Nell'ambito della programmazione orientata agli oggetti, chi usa un certo oggetto non sempre è interessato a conoscere come è implementato. Molto spesso l'oggetto è stato costruito da un programmatore, diverso da colui che lo userà. A quest'ultimo interessa sapere, invece, come interagire con l'oggetto, quali sono i metodi che mette a disposizione e come richiamarli, quali sono gli attributi pubblici.

Potremmo fare un parallelismo del tipo: il programmatore è come il meccanico che conosce i dettagli del funzionamento dell'automobile, mentre il pilota come utilizzare l'automobile, e può ignorare i dettagli del funzionamento dell'automobile.

Questo modo di intendere gli oggetti induce a considerare un oggetto come una scatola nera (**blackbox**). I dettagli sulle caratteristiche e le strutture dell'oggetto sono nascosti all'interno, garantendo l'information hiding (mascheramento dell'informazione). Il vantaggio introdotto dall'information hiding consiste nel nascondere la parte implementativa dell'oggetto, cioè nascondere il modo con cui i metodi sono realizzati, basta conoscere come si utilizzano. Questo è quello che facciamo quando utilizziamo le librerie di classi offerte dal framework .NET: impariamo ad utilizzarle ma non sappiamo come sono implementati.

# Information hiding

Si parla di information hiding o occultamento delle informazioni per indicare anche il caso in cui alcuni attributi e metodi possono essere nascosti all'esterno dell'oggetto, cioè resi invisibili agli altri oggetti. Gli attributi e metodi nascosti (o invisibili) sono detti privati. Gli attributi e metodi visibili sono detti pubblici e costituiscono l'**interfaccia** dell'oggetto.

Ogni oggetto ha una porzione esterna o interfaccia e una porzione interna occultata (con visibilità limitata). **Gli altri oggetti comunicano solo attraverso l'interfaccia.** La parte interna è protetta da manomissioni e inoltre può essere modificata senza influire su altre parti del programma.

L'insieme dei metodi, che consentono l'interazione con l'oggetto, rappresenta la sua interfaccia e lo separa dal mondo esterno. Chi utilizza l'oggetto deve conoscere solo la sua interfaccia. In questo modo può sapere quali metodi possono essere invocati, quali sono i parametri da passare e qual è il tipo del valore di ritorno.

Nella pratica della programmazione ad oggetti, solitamente, si procede nella definizione dell'interfaccia degli oggetti, in modo da concentrare la prima fase della progettazione del sistema agli oggetti coinvolti, alle loro relazioni, interazioni e, successivamente, quando la progettazione del sistema è completa, e risponde alle esigenze emerse in fase di analisi, si procede all'implementazione dei metodi.

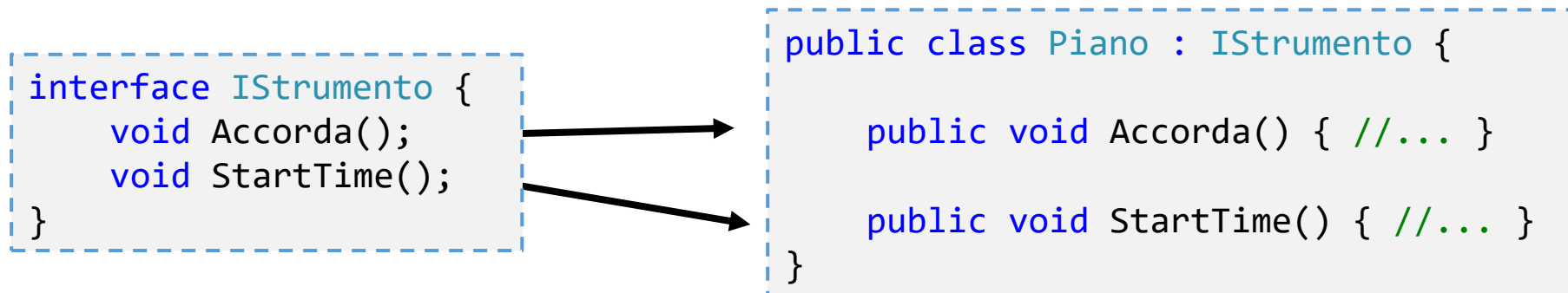
# Incapsulamento e Information hiding

In estrema sintesi l'incapsulamento e l'information hiding è un meccanismo di controllo:

- I dati e i dettagli implementativi sono invisibili all'utilizzatore
- I dati e i dettagli implementativi sono incapsulati
- l'accesso ai dati avviene attraverso un'interfaccia (funzioni definite)
- tutti gli attributi dovrebbero essere privati

# Interfacce

- Un'interfaccia è simile ad una **classe astratta pura**.
- Un'interfaccia è priva di qualsiasi implementazione e di modificatore di accessibilità (**public**, **private**, ecc.).
- Un'interfaccia definisce un contratto che la classe che la implementa deve rispettare.
- Una classe può implementare più interfacce contemporaneamente.





# Perché usare le interfacce?

- Perché rappresentano tipi astratti che permettono di limitare l'accoppiamento.
- Perché in .NET l'ereditarietà multipla non è ammessa.
- Per creare dipendenze tra i tipi senza ricorrere all'ereditarietà.
- Per caricare assembly in modo dinamico e realizzare un sistema pluggabile.

# Convenzioni sul codice

- **Notazione ungherese:** al nome dell'identificatore viene aggiunto un prefisso che ne indica il tipo (es. `intNumber` identifica una variabile intera)
- **Notazione Pascal:** l'inizio di ogni parola che compone il nome dell'identificatore è maiuscola, mentre tutte le altre lettere sono minuscule (es. `FullName`)
- **Notazione Camel:** come la notazione Pascal, a differenza del fatto che la prima iniziale deve essere minuscola (es. `fullName`)

# Convenzioni sul codice

Elemento/i	Notazione
Namespace	Notazione Pascal
Classi	Notazione Pascal
Interfacce	Notazione Pascal
Strutture	Notazione Pascal
Enumerazioni	Notazione Pascal
Campi privati	Notazione Camel, eventualmente preceduta dal carattere di sottolineatura (esempio: <code>_fullName</code> )
Proprietà, metodi ed eventi	Notazione Pascal Parametri dei metodi e delle funzioni in
generale	Notazione Camel
Variabili locali	Notazione Camel

# Esercitazione: Azienda

In un'azienda lavorano Tecnici, Stagisti e Manager.

Tecnico: Nome, Cognome, Paga Oraria, Codice Fiscale

Stagista: Nome, Cognome, Compenso Mensile fisso

Manager: è un Tecnico. Ha un bonus Mensile in più rispetto al tecnico.

Per ogni dipendente deve essere possibile stampare i dati anagrafici (nome e cognome), il ruolo (se si tratta di Stagista, Tecnico o Manager), Stampare lo stipendio, stampare le ferie.

# Esercitazione Calcio

Gli atleti hanno nome, cognome, età

Calciatori hanno ruolo e numero maglia (ruoli = centrocampista e difensore portiere e attaccante)

Portieri hanno di default il numero maglia = 1 il numero gol subito  
// gli attaccanti hanno il numero gol fatti a partita // Una  
squadra di calcio è formata da 11 calciatori di cui // un portiere  
// 4 difensori // 4 centrocampisti // 2 attaccanti // Per  
svolgere una partita serve anche un arbitro (l'arbitro è un atleta)

# Esercitazione: Forme geometriche

Realizzare un'interfaccia Forma che rappresenti una forma geometrica, e che preveda dei metodi per inserire le dimensioni da tastiera, stampare a video le dimensioni, calcolare il perimetro e calcolare l'area.

Implementare tale interfaccia nelle classi concrete Rettangolo, Triangolo e Cerchio, e poi derivare dalla classe Rettangolo la classe Quadrato.

Realizzare poi un'app console che permetta all'utente di inserire degli oggetti di tipo Forma da tastiera, e di stampare sullo schermo le dimensioni, area e perimetro di tutte le forme inserite.

# ADO.NET



**Alice Colella**

Junior Developer@icubedsrl

[Alice.Colella@icubed.it](mailto:Alice.Colella@icubed.it)



# ADO.NET

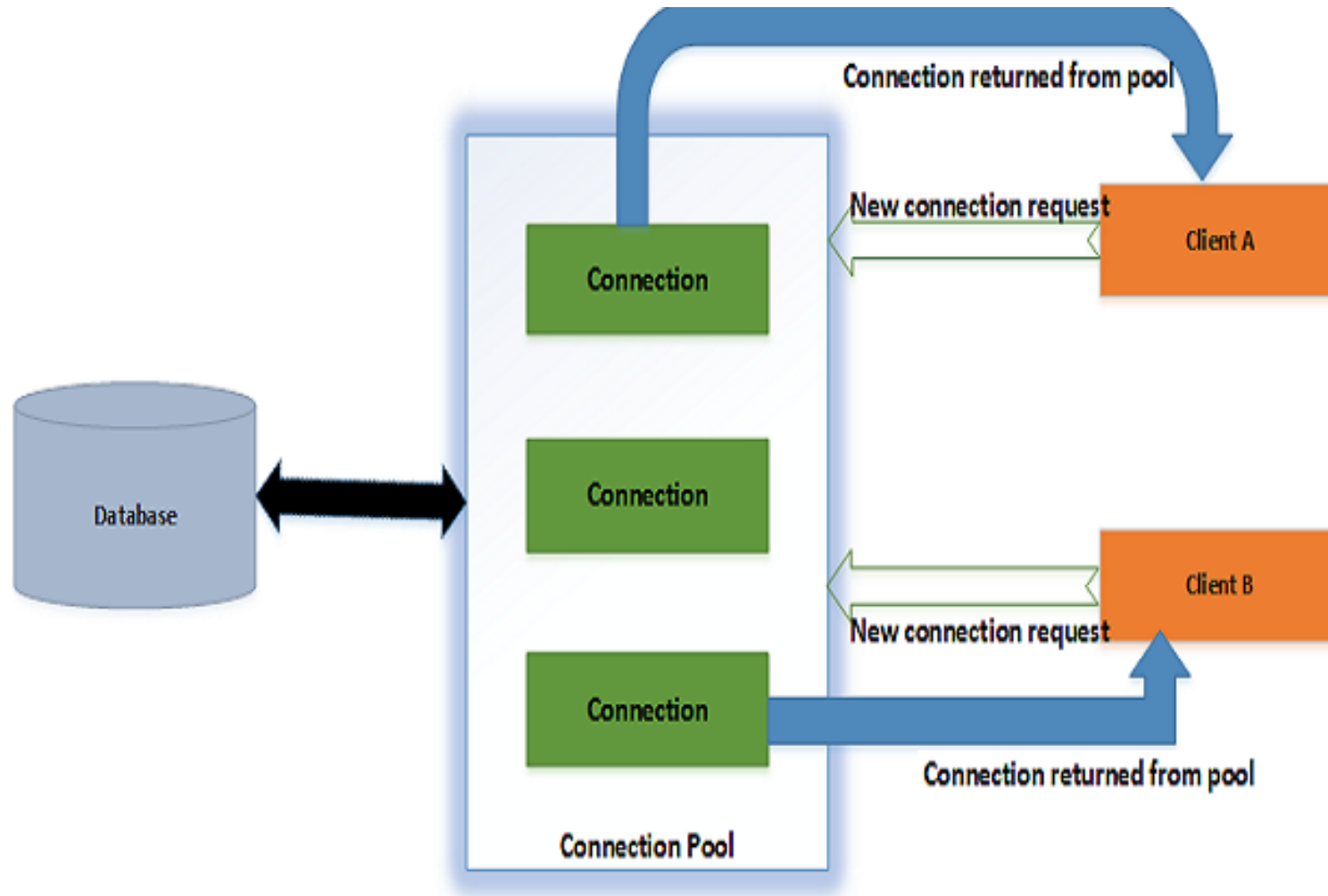
- **ADO.NET** è una tecnologia di accesso ai dati di Microsoft
- È costituito da un insieme di componenti software che i programmatori possono utilizzare per accedere ai dati e ai servizi dati da un database
- Fornisce un accesso coerente
  - alle origini dati come SQL Server
  - alle origini dati esposte tramite OLE DB e ODBC



# ADO.NET - Provider

- ADO.NET include i **Provider** di dati per
  - la connessione a un database
  - l'esecuzione di comandi
  - il recupero dei risultati
- I **provider** di dati .NET Framework forniscono una serie di oggetti per gestire l'accesso ai dati
  - L'oggetto **Connection** fornisce connettività a un'origine dati tramite una Connection String
  - L'oggetto **Command** consente l'accesso ai comandi del database per restituire dati, modificare dati, eseguire procedure memorizzate e inviare o recuperare informazioni sui parametri

# ADO.NET - Connection Pool



# ADO.NET - Connection Pool

- Ogni volta che un utente chiama `OPEN()` su una connessione, il pooler cerca una connessione disponibile nel pool
  - Se è disponibile una connessione in pool, la restituisce al chiamante invece di aprire una nuova connessione.
- Quando l'applicazione chiama `CLOSE()` sulla connessione, il pooler la restituisce al set di connessioni attive in pool anziché chiuderlo
- Una volta che la connessione viene restituita al pool, è pronta per essere riutilizzata alla successiva chiamata `Open`

# ADO.NET – Connection String

- Una stringa di connessione contiene le informazioni di inizializzazione fondamentali per creare una connessione con un database.

```
Server=tcp:platone.database.windows.net,1433; Initial Catalog=University; Persist  
Security Info=False; User ID={your user}; Password={your_password};  
MultipleActiveResultSets=False; Encrypt=True; TrustServerCertificate=False;  
Connection Timeout=30;
```

# ADO.NET - Modalità

- **ADO.NET** consente l'accesso ai dati in due modalità distinte:
- **Connected Mode**  
(*Connection, Commands, DataReader ...*)
- **Disconnected Mode**
  - (*Connection, Adapter, DataSet, DataTable ...*)

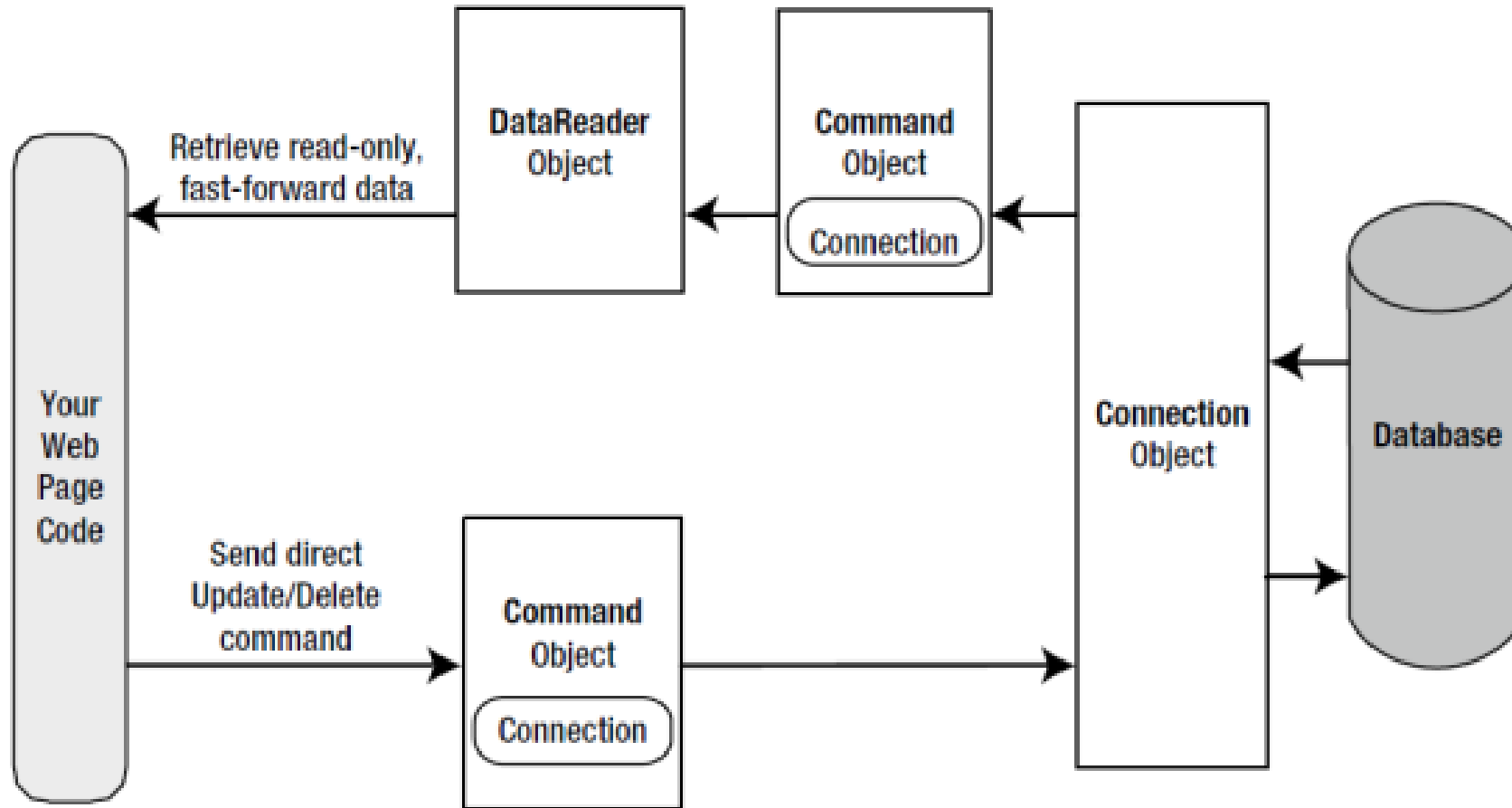
# ADO.NET – Connected Mode

- Il **Connected Mode** fornisce
  - accesso di sola lettura (e forward-only) ai dati nell'origine dati
  - la possibilità di eseguire comandi sull'origine dati

# ADO.NET – Connected Mode

- Principali classi utilizzate in Connected Mode:
- Connection (*SqlConnection*)
- Command (*SqlCommand*)
- DataReader (*SqlDataReader*)
- Parameter (*SqlParameter*)

# ADO.NET – Connected Mode





# ADO.NET – Connected Mode

1. Creare Connessione
2. Aprire Connessione
3. Creare Command
  1. Creare Parametri se necessario
4. Eseguire Command (DataReader/NonQuery/Scalar)
5. Lettura Dati a schermo
6. Chiudere Connessione

# Demo



Connected Mode

