

Parte a Quiz. Cognome e Nome: _____

Rispondere alle seguenti domande a risposta multipla. Fate attenzione che

- alcune [combinazioni di] risposte sono così sbagliate da portare *punteggio negativo*;
- alcune domande hanno più risposte corrette e per totalizzare il punteggio pieno dovete selezionarle *tutte*.

Segnare con **Y** le affermazioni vere, con **N** quelle false.

Esercizio 3

vincolo Where ==> I Enumerable

Data l'espressione LINQ

```
new []{26.85, 28.86, 29.1, 30.94, 31.02, 31.4, 35.61, 38.6, 39.79}.  
Where(i => i >= 30);
```

quali delle seguenti affermazioni sono vere?

- Il tipo dell'espressione è `IEnumerable<double>`
- Il tipo dell'espressione è `double[]`
- Il tipo dell'espressione è `IQueryable<double>`
- Il tipo dell'espressione è `double`
- L'espressione non è sintatticamente corretta perché il metodo `Where` si può invocare solo su un'espressione di tipo `DbSet<T>`
- L'espressione può essere assegnata ad una variabile di tipo `IQueryable<double>`
- Per poter assegnare l'espressione ad una variabile di tipo `IQueryable<double>` bisogna prima cambiarle il tipo, ad esempio usando il metodo `AsQueryable()`
- L'espressione può essere assegnata ad una variabile di tipo `double[]`
- All'espressione può essere assegnata un valore di tipo `double[]`
- Nessuna delle precedenti affermazioni è vera

Esercizio 4

Si consideri il seguente frammento di codice basato sull'entity framework.

```
public class Alpha {
    public int AlphaId { get; set; }
    public int BetaId { get; set; }
    public virtual ICollection<Beta> Betas { get; set; }
    public virtual ICollection<Alpha> Alphas { get; set; }
}
public class Beta {
    public int BetaId { get; set; }
    [MaxLength(50), Index(IsUnique = true)]
    public string Beta1 { get; set; }
    public virtual ICollection<Alpha> Alphas { get; set; }
}
public class MyContext : DbContext {
    public DbSet<Alpha> Alphas { get; set; }
    public DbSet<Beta> Betas { get; set; }
    /*... Constructors...*/
}
```



L'entità **Alpha** è collegata all'entità **Beta** *esclusivamente* da una relazione molti a molti, rappresentata dalla coppia di proprietà **Alphas** (nella classe **Beta**) e **Betas** (nella classe **Alpha**)



L'entità **Alpha** è collegata all'entità **Beta** *esclusivamente* da due relazione uno a molti, rappresentate rispettivamente dalle proprietà **Alphas** (nella classe **Beta**) e **Betas** (nella classe **Alpha**)



La proprietà **BetaId** nella classe **Alpha** è una proprietà di navigazione verso l'entità **Beta**, cioè nella tabella generata sarà una chiave esterna verso la tabella che rappresenta l'entità **Beta**



la proprietà **Alphas** nella classe **Alpha** rappresenta una auto-relazione sull'entità **Alpha**



la proprietà **Alphas** nella classe **Alpha** è equivalente alla proprietà **Alphas** del contesto, nel senso che entrambe restituiscono sempre tutti gli oggetti di tipo **Alpha** noti



la base di dati generata dall'entity framework per questo frammento contiene le due tabelle per **Alphas** e **Betas** e nessun'altra



la base di dati generata dall'entity framework per questo frammento contiene le due tabelle per **Alphas** e **Betas** e una per la relazione molti a molti fra **Alpha** e **Beta** e nessun'altra



la base di dati generata dall'entity framework per questo frammento contiene più di tre tabelle.



nella base di dati generata dall'entity framework per questo frammento, la tabella per **Alphas** ha un'unica chiave esterna verso se stessa.



Il seguente frammento di codice può sollevare un'eccezione dovuta a violazione di chiave (secondaria)?

```
using (var c = new EntityUnderstanding.MyContext(ConnectionString)) {
    if (c.Betas.Any(b=>b.Beta1=="Paperino"))
        throw new ApplicationException("name already in use");
    var beta = c.Betas.Create();
    beta.Beta1 = "Paperino";
    c.Betas.Add(beta);
    c.SaveChanges();
}
```

Esercizio 3 (4 punti)

Dato il seguente frammento di codice, quali test avranno successo?

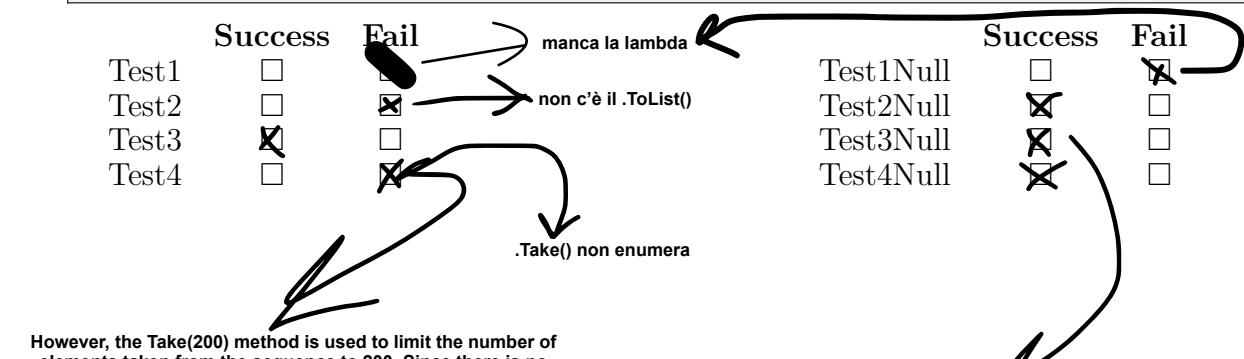
```

public static IEnumerable<double> M(this IEnumerable<int> s) {
    if (null==s) throw new ArgumentNullException();
    return PrivateM();

    IEnumerable<double> PrivateM() {
        foreach (var n in s) {
            if (0 == n) throw new ArgumentException();
            yield return 1.0 / n;
        }
    }
}

[TestFixture]
public class Test {
    IEnumerable<int> S() {
        var i = -123;
        while (true) yield return i++;
    }
    IEnumerable<int> Null() { return null; }
    [Test]
    public void Test1()
    { Assert.That(S().M(), Throws.TypeOf<ArgumentException>()); }
    [Test]
    public void Test2()
    { Assert.That(()=>S().M(), Throws.TypeOf<ArgumentException>()); }
    [Test]
    public void Test3()
    { Assert.That(()=>S().M().ToArray(), Throws.TypeOf<ArgumentException>()); }
    [Test]
    public void Test4()
    { Assert.That(() => S().M().Take(200), Throws.TypeOf<ArgumentException>()); }
    [Test]
    public void Test1Null()
    { Assert.That(Null().M(), Throws.TypeOf<ArgumentNullException>()); }
    [Test]
    public void Test2Null()
    { Assert.That(() => Null().M(), Throws.TypeOf<ArgumentNullException>()); }
    [Test]
    public void Test3Null()
    { Assert.That(()=>Null().M().ToArray(), Throws.TypeOf<ArgumentNullException>()); }
    [Test]
    public void Test4Null()
    { Assert.That(()=> Null().M().Take(200), Throws.TypeOf<ArgumentNullException>()); }
}

```



However, the Take(200) method is used to limit the number of elements taken from the sequence to 200. Since there is no condition to stop the infinite sequence in the S() method, the Take(200) will keep trying to take elements indefinitely, resulting in the test case not completing and eventually timing out.

effettivamente ritornano tutti null

Esercizio 4 (4 punti)

Supponendo che le seguenti classi siano gestite usando l'Entity Framework, indicare se le affermazioni seguenti sono vere o false.

```
public class Student {
    public int StudentId { get; set; }
    [Required]
    [Index("theIndex", Order=1, IsUnique = true)]
    public string StudentName { get; set; }
    [Index("theIndex", Order = 2, IsUnique = true)]
    public DateTime? DateOfBirth { get; set; }
    public byte[] Photo { get; set; }
    public decimal Height { get; set; }
    [Required]
    public float Weight { get; set; }

    public Grade Grade { get; set; }
    public int TutorId { get; set; }
}

public class Grade { public int GradeId { get; set; } /*...*/ }
public class Tutor { public int TutorId { get; set; } /*...*/ }
```

Vero Falso

- l'attribute **Required** per la property **StudentName** è superfluo, perché è una chiave quindi non nullable
- l'attribute **Required** per la property **StudentName** rende non nullable la colonna corrispondente nel DB
- l'attribute **Required** per la property **StudentName** genera la verifica che il valore non sia nullo durante la chiamata di **SaveChanges** prima di effettuare la connessione al DB
- l'attribute **Required** per la property **Weight** è superfluo, perché il tipo **float** non è nullable → **i tipi valore non sono nullabili di default, hanno bisogno del "?"**
- l'attribute **Required** per la property **Weight** è indispensabile, perché il tipo **float** non è nullable, se no si avrebbe un conflitto con il default del DB in cui le colonne sono nullabili. Una valida alternativa per evitare il conflitto sarebbe dichiarare **Weight** con tipo **float?**. Ma se si lascia l'inconsistenza si ottiene un errore di compilazione.
- la property **Grade** rappresenta una proprietà di navigazione verso la classe **Grade**
- affinché la property **Grade** rappresenti una proprietà di navigazione verso la classe **Grade** è indispensabile aggiungere anche la property **public int GradeId** → **non è un requisito assoluto**
- la property **Grade** rappresenta il lato uno di una associazione uno a molti verso la classe **Grade** qualunque sia il codice di **Grade** → **credo sia un'associazione (1,1)**
- se la classe **Grade** non contiene nessuna property di navigazione verso la classe **Student**, allora la property **Grade** rappresenta il lato uno di una associazione uno a molti verso la classe **Grade** →
- la property **TutorId** rappresenta una chiave esterna che riferisce a **Tutor**
- la property **TutorId** è un intero che nulla ha a che vedere con la classe **Tutor**

Per stabilire
un'associazione uno-a-
molti tra due classi,
entrambe le classi
devono avere una
relazione tra di loro, di
solito attraverso una
property di navigazione

Esercizio 3 (9 punti)

Per ciascuna delle seguenti affermazioni, indicate se è vera o falsa

1. In C#, nelle intestazioni dei metodi, le eccezioni sollevate

Vero Falso

- Non si devono dichiarare
- Si devono dichiarare *tutte* con throws
- Si devono dichiarare con throws *solo* quelle *user defined*

2. Se un oggetto di classe C ha bisogno di un logger di tipo L, secondo la dependency-injection:

Vero Falso

- C deve usare la reflection per ottenere un'istanza di L
- L deve fornire un costruttore senza parametri, in modo che C possa fare `new L()`
- I costruttori di C devono avere un parametro di tipo L

3. L'esecuzione del comando `using (T x=e) { ... }` corrisponde grosso modo a quella di:

Vero Falso

- T x=e; `try { ... } finally { x.Dispose(); Delete(x); }`
- `try {...} finally { x.Dispose(); Delete(x); }`
- T x=e; `try { ... } finally { x.Dispose(); }`

4. Se x è un `IQueryable`, le seguenti espressioni si possono usare come `IEnumerable`

Vero Falso

- x
- x.AsEnumerable()
- `new IEnumerable(x)`

5. In Git i seguenti comandi richiedono la connessione al server:

Vero Falso

- git log
- git clone
- git commit

6. Nel testing, si usano

Vero Falso

- Gli stub per lo state-based testing, i mock per l'interaction-based testing
- I mock per lo state-based testing, gli stub per l'interaction-based testing
- Indifferentemente, stub e mock (che sono fra loro sinonimi)

7. Per definire un custom-attribute si deve

Vero Falso

- Scrivere un file XML
- Usare l'ADO Entity Framework
- Scrivere una classe

8. Per il passaggio di parametri per riferimento in C#

Vero Falso

- Si usa la keyword `ref` sia nella dichiarazione del parametro, sia nella chiamata
- Si usa la keyword `ref` solo nella dichiarazione del parametro
- Si usa la keyword `ref` solo quando si passa l'argomento al momento della chiamata

9. In uno unit-test, ci aspettiamo che:

Vero Falso

- Ci sia un'unica asserzione, alla fine del metodo
- Ci siano tante asserzioni, una per ogni proprietà verificata dal test
- Ci sia un'asserzione dopo ogni istruzione in modo da tracciare dove fallisce

Esercizio 3 (9 punti)

Per ciascuna delle seguenti affermazioni, indicate se è vera o falsa

1. In C#, se un metodo solleva una eccezione di tipo E

Vero Falso

- lo stesso metodo deve anche catturarla e gestirla
- nell'intestazione del metodo E deve comparire all'interno della clausola `throws`
- lo stesso metodo non può sollevare altre eccezioni di tipo E

2. Nell'istante in cui un oggetto di tipo C con accesso esclusivo a un file diventa irraggiungibile

Vero Falso

- il file viene immediatamente rilasciato automaticamente
- se C non implementa `IDisposable` si ha errore dinamico
- anche se C implementa `IDisposable` il file potrebbe restare "lockato"

3. Se un oggetto di classe C ha bisogno di un logger di tipo L, secondo la dependency-injection:

Vero Falso

- deve esistere anche la classe factory per L da usare in C
- il DI container intercetterà `new L()` nei costruttori di C e materializzerà il logger
- i costruttori di C devono avere un parametro di tipo L

4. Per poter passare come parametro un metodo, il tipo del parametro può essere

Vero Falso

- `Func<..>, Action<..>` o altro tipo *delegate*
- un tipo *lambda*
- `FunctionPointer<...>` o altro tipo puntatore

5. Per definire un custom-attribute

Vero Falso

- bisogna essere in un progetto per la piattaforma .NET Standard Library
- si deve avere un riferimento a `System.Annotations.CustomAttributes`
- basta estendere la classe `CustomAttribute`

6. Considerando i vari tipi di passaggio di parametri in C# 9

Vero Falso

- per dichiarare un singolo parametro si possono usare simultaneamente `ref` e `in`
- dichiarare un parametro come `out` impone vincoli statici sul corpo del metodo
- usare `ref` impone vincoli statici sui parametri attuali usabili nella chiamata

7. Se in uno unit-test compaiono più asserzioni a livello di annidamento top, cioè non contenute in altri statement/espressioni:

Vero Falso

- il test runner solleva un'eccezione (uno unit test non può contenere più asserzioni)
- alla prima asserzione fallita il test termina l'esecuzione
- il test runner indica tutte le asserzioni fallite

8. Anche se il servizio di riferimento per un repo (l'*origin* del repo) non è raggiungibile, i seguenti comandi possono ugualmente avere successo:

Vero Falso

- `git pull`
- `git push`
- `git branch [branch-name]`

9. Confrontando le interfacce `IQueryable` e `IEnumerable`

// **IQueryable deriva da IEnumerable**

Vero Falso

- `IQueryable` ha più membri di `IEnumerable`
- `IEnumerable` ha più membri di `IQueryable`
- `IEnumerable` ha gli stessi membri di `IQueryable`, ma con diverse implementazioni

Esercizio 3 (9 punti)

Per ciascuna delle seguenti affermazioni, indicate se è vera o falsa

1. In un test del metodo statico M() della classe C:

```
public static IEnumerable<int> M(){
    for (int i = 0; i < 10; i++) yield return i;
    throw new InvalidOperationException();
}
```

Le seguenti asserzioni sono *successful*

Vero Falso

- Assert.That(C.M(), Is.Not.Empty);
- Assert.That(C.M(), Throws.TypeOf<InvalidOperationException>());
- Assert.That(()=>C.M(), Throws.TypeOf<InvalidOperationException>());
- Assert.That(()=>C.M().Any(), Throws.TypeOf<InvalidOperationException>());
- Assert.That(()=>C.M().ToArray(), Throws.TypeOf<InvalidOperationException>());
- Assert.That(C.M().ToArray(), Throws.TypeOf<InvalidOperationException>());

2. Se le interfacce I1 e I2 definiscono lo stesso metodo M() fornendo diverse implementazioni di default

Vero Falso

- nessuna classe può implementare simultaneamente I1 e I2
- nella definizione di una classe che implementa simultaneamente I1 e I2 bisogna optare per una delle due implementazioni
- una classe che implementa simultaneamente I1 e I2 può implementare in modo esplicito le due varianti di M()

3. Dato

```
IEnumerable<string?> A = ....;
var a1 = A.Skip(3).FirstOrDefault();
var a2 = A.ElementAtOrDefault(3);
var b = A.Take(3);
var a3 = A.FirstOrDefault();
```

Vero Falso

- qualunque sia il valore di A si ha a1==a2
- si ha a1==a2 solo se A ha almeno quattro elementi → testati :)
- se non ci sono accessi concorrenti ad A, a1==a3 qualsiasi siano gli elementi di A

4. Nell'ambito dell'Entity Framework Core, la scelta del tipo di base di dati da usare per rendere permanenti le entità può essere effettuato

Vero Falso

- nel costruttore del contesto
- nel metodo OnModelCreating
- nel metodo SaveChanges

Anche se la chiamata a Any() restituisce true, ciò indica che l'IQueryable<int> X contiene almeno un elemento. Tuttavia, la chiamata a X.First() può sollevare un'eccezione se l'IQueryable<int> non contiene alcun elemento. In tal caso, la chiamata a First() provocherà un'eccezione di tipo InvalidOperationException indicando che la sequenza è vuota.

5. Dato IQueryable<int> X

Vero Falso

- in X.Any()&&32==X.First() la chiamata a First può sollevare eccezioni anche se quella a Any() restituisce true
- enumerazioni diverse di X possono produrre risultati differenti
- in foreach(var i in X){...} var b=X.Any() a b viene sempre assegnato false perché X è stato visitato fino alla fine e non ci sono ulteriori elementi

Quindi, anche se la condizione di Any() è soddisfatta e restituisce true, è ancora possibile che la chiamata a First() sollevi un'eccezione se non ci sono elementi nell'IQueryable<int>.

Nel contesto di un foreach loop, la chiamata a X.Any() non influenza sul valore di b all'interno del loop. La chiamata a X.Any() restituisce un valore booleano che indica se l'IQueryable<int> X contiene almeno un elemento o meno, ma non modifica il flusso del foreach loop.

Durante l'esecuzione del loop, ogni elemento dell'IQueryable<int> viene visitato uno alla volta, e il corpo del loop viene eseguito per ciascun elemento. La condizione di X.Any() viene valutata solo una volta all'inizio del loop e non viene influenzata dai successivi passaggi del loop.

Quindi, il valore di b sarà true se l'IQueryable<int> X contiene almeno un elemento, indipendentemente da quanti elementi vengono visitati nel loop.

6. In Git, il comando `rebase`

Vero Falso

- può modificare la storia passata del repository
- causa errore in esecuzione se si è precedentemente fatto `push` del repo sul server
- causa errore in esecuzione se un diverso utente ha già fatto `pull` dei commit coinvolti

7. Considerate queste due varianti (i puntini indicano parti non rilevanti ai fini dell'esercizio) dal punto di vista della DI (Dependency Injection)

```
class D { ... }
class C {
    C() { ... }
    void M() {
        ... var x = new D(); ...
    }
}
```

```
class D: ID { }
class C {
    private ID _myD;
    C(ID d) { _myD = d; ... }
    void M() {
        ... var x = _myD; ...
    }
}
```

1

2

Vero Falso

- la variante 1 rispetta i principi della DI
- la variante 2 rispetta i principi della DI
- non ci può essere differenza fra il comportamento delle varianti 1 e 2
- non si può applicare la DI alla variante 1 senza alterare il comportamento di `M`
- per applicare la DI alla variante 1 il costruttore di `C` dovrebbe prendere come parametro una factory per `ID`
- quando si aggiungono parametri al costruttore per applicare la DI è opportuno mantenere anche il costruttore originale, per evitare di rompere i client

quando si aggiungono parametri al costruttore per applicare la Dependency Injection, è
opportuno mantenere anche il costruttore originale per evitare di rompere i client esistenti che
potrebbero dipendere dal costruttore senza parametri. In questo modo si garantisce la
retrocompatibilità del codice esistente

Esercizio 3 (9 punti)

Per ciascuna delle seguenti affermazioni, indicate se è vera o falsa

1. Si consideri il seguente frammento di un progetto che usa l'Entity Framework

```
public class A {
    public int AId { get; set; }
    public int BId { get; set; }
    public List<B> MyBs { get; set; }
}
[Index(nameof(X),nameof(Y),IsUnique = true)]
public class B {
    public int BId { get; set; }
    public int X { get; set; }
    public int Y { get; set; }
    public List<A> MyAs { get; set; }
}
public class C {
    public int CId { get; set; }
    public int AId { get; set; }
    public A A { get; set; }
}
public class MyDbContext : DbContext {
    public DbSet<A> As { get; set; }
    protected override void OnConfiguring(DbContextOptionsBuilder options) { ... }
}
```

Vero Falso

- l'EF non può identificare entità in questo frammento perché nessuna classe estende la classe base `Entity`, radice per l'EF di tutte le entità
- la classe `A` è un'entità, ovvero ha una tabella corrispondente sul DB
- la classe `B` è un'entità, ovvero ha una tabella corrispondente sul DB
- la classe `C` è un'entità, ovvero ha una tabella corrispondente sul DB
- per rappresentare questo frammento, l'EF genera meno di 3 tabelle sul DB → **ne genera 3**
- per rappresentare questo frammento, l'EF genera esattamente 3 tabelle sul DB
- per rappresentare questo frammento, l'EF genera più di 3 tabelle sul DB
- `MyDbContext` non è staticamente corretto perché non ha property di tipo `DbSet` e di tipo `DbSet<C>`
- l'EF non può identificare entità in questo frammento perché nessuna classe è annotata con l'attribute `EntityTypeAttribute`
- la definizione della classe `MyDbContext` è indispensabile per determinare quali sono le entità
- la property `BId` nella classe `A` è una proprietà di navigazione verso `B`
- le property `MyBs` in `A` e `MyAs` in `B` rappresentano i due lati di una relazione molti-a-molti per l'EF
- le property `MyBs` in `A` e `MyAs` in `B` rappresentano due relazioni uno-a-molti distinte per l'EF
- il codice `var b1= new B() { X = 1, Y = 1 }; var b2= new B() { X = 1, Y = 1 };` solleva eccezione per violazione di indice `unique`
- nella tabella associata a `B` ci possono essere più righe che differiscono solo per il valore della chiave primaria
- l'EF interpreta la property `AId` nella classe `C` come la chiave della proprietà di navigazione `A`
- l'EF interpreta la property `BId` nella classe `A` come la chiave della proprietà di navigazione `MyBs`
- le classi `A`, `B` e `C` sono dinamicamente scorrette perché non hanno un costruttore

2. Si consideri il seguente frammento di codice

```
public static class C {
    public static int F(int x, int y) {
        if(x>y) return x;
        if (y > x) return y;
        throw new ArgumentException(message:"", paramName:nameof(x));
    }
    public static int G(int x, int y) {
        if (x > y) return x;
        if (y > x) return y;
        throw new ArgumentException(message: nameof(x));
    }
}
[TestFixture]
public class Test {
    [Test]
    public void TF1() {
        Assert.That(()=>C.F(3,3), Throws.TypeOf<ArgumentException>()
            .With.Property("ParamName").EqualTo("x"));
    }
    [Test]
    public void TF2() {
        Assert.That(() => C.F(3, 3), Throws.InstanceOf<Exception>());
    }
    [Test]
    public void TF3() {
        Assert.That(() => C.F(3, 3), Throws.TypeOf<Exception>());
    }
    [Test]
    public void TG1([Random(10,50,3)] int x, [Random(51, 100, 3)] int y) {
        Assert.That(C.G(x, y), Is.EqualTo(y));
    }
    [Test]
    public void TG2() {
        Assert.That(() => C.G(3, 3), Throws.TypeOf<ArgumentException>()
            .With.Property("ParamName").EqualTo("x"));
    }
    [Test]
    public void TG3() {
        var result = C.G(3, 3);
        Assert.That(result, Throws.InstanceOf<ArgumentException>());
    }
}
```

Vero Falso

- Il test TF1 ha successo
- Il test TF1 non è corretto perché usa metodi inesistenti
- Il test TF2 ha successo
- Il test TF3 ha successo
- Il test TG1 ha successo
- Il test TG1 è un test parametrico che viene tradotto da NUnit in 9 test base
- Il test TG1 non è corretto, perché l'attributo Random va usato all'interno dell'attributo TestCase e non sui parametri
- Il test TG2 ha successo
- Il test TG3 ha successo → manca la lambda

Esercizio 3 (9 punti)

Per ciascuna delle seguenti affermazioni, indicate se è vera o falsa

1. Nell'ambito dei custom attribute

Vero Falso

- ➔ i custom attribute non esistono più nelle versioni più recenti del .NET Framework, a partire da .NET 5.0.11 inclusa
- ➔ si possono definire solo in progetti per la piattaforma .NET Core
- ➔ devono avere property o metodi pubblici, altrimenti non servono a nulla
- ➔ devono avere solo il costruttore di default altrimenti non sono staticamente corretti
- ➔ una qualsiasi classe che estenda `System.Attribute` è un custom attribute
- ➔ non sono estendibili, ovvero se una classe rappresenta un custom attribute non se ne può definire una specializzazione
- ➔ una classe può avere (essere decorata da) al più un custom attribute
- ➔ si può restringere il campo di applicazione di un custom attribute a solo alcuni elementi del linguaggio (classi, metodi...)

2. Supponete di avere un oggetto o di tipo C con accesso esclusivo a xyz.mp4

Vero Falso

- ➔ appena o diventa irraggiungibile il garbage collector provvede a rilasciare il lock su xyz.mp4
- ➔ appena o diventa irraggiungibile il garbage collector provvede a rilasciare lo spazio disco occupato da xyz.mp4
- ➔ se o non viene usato in un costrutto `using` si ha errore statico
- ➔ solo il costrutto `using` garantisce che il lock su xyz.mp4 sia rilasciato correttamente
- ➔ per essere staticamente corretta C deve implementare `IDisposable`
- ➔ solo se C implementa `IDisposable` si può usare il costrutto `using` per oggetti di tipo C

3. Nell'ambito dei Versioning System e più specificamente di Git

Vero Falso

- ➔ ogni commit su Git ha un singolo parent
- ➔ un commit su Git può avere un numero arbitrario di figli, in particolare nessuno o più di uno
- ➔ Git è basato sul modello client-server
- ➔ Fare pull genera conflitti sui remote su cui si fa il push
- ➔ Git prevede un sistema di garbage collection che fa pulizia dei commit diventati irraggiungibili
- ➔ la versione di un file di cui si è fatto staging è quella che verrà aggiunta al prossimo commit anche se fra staging e commit il file viene modificato

4. Volendo definire una variabile myDbSet che rappresenta in memoria il risultato di una query effettuata usando l'EF.

Vero Falso

- ➔ Una doppia enumerazione di myDbSet genera errore dinamico
- ➔ Accedendo due volte all'i-esimo elemento di myDbSet si possono ottenere valori diversi
- ➔ Scegliere `List<C>`, dove C è il tipo degli elementi risultanti dalla query, come tipo di myDbSet può richiedere più RAM di quella disponibile
- ➔ Il tipo di myDbSet deve implementare `IDbSet`, altrimenti si ha un errore statico
- ➔ Il tipo più naturale per myDbSet è `IQueryable` o un suo tipo derivato
- ➔ Accedere direttamente (usando un indice o il metodo `ElementAt()`) a più elementi di myDbSet causa multiple enumerazioni
- ➔ Ogni modifica a elementi di myDbSet viene automaticamente e immediatamente riflessa sul corrispondente elemento nella base di dati

Cognome e nome:

Esercizio 3 (9 punti)

Per ciascuna delle seguenti affermazioni, indicate se è vera o falsa

1. Anche se il server Git di riferimento per un repo (l'*origin* del repo) non è raggiungibile, i seguenti comandi possono ugualmente avere successo:

Vero Falso

- git commit
- git rebase
- git pull

2. Confrontando le interfacce `IQueryable` e `IEnumerable`

Vero Falso

- se un metodo si può chiamare su un `IEnumerable` si può chiamare anche su un `IQueryable`
- se un metodo si può chiamare su un `IQueryable` si può chiamare anche su un `IEnumerable`
- un `IQueryable` ha il metodo `GetEnumerator`

non è detto... `IQueryable` estende `IEnumerable`

3. Date le dichiarazioni

```
public class Parent { public void M(){Console.WriteLine("Parent");}}
public class Child: Parent { public void M() {Console.WriteLine("Child");}}
```

Vero Falso

- Il codice Parent o = new Child(); o.M(); genera un errore statico
- Il codice Parent o = new Child(); o.M(); stampa Parent
- Il codice Parent o = new Child(); o.M(); stampa Child

eseguito online :

4. Data la dichiarazione `public class C<T>{/*...*/}`

Vero Falso

- C<3> c; causa errore statico
- C<3> c; causa errore dinamico
- Se A è sottoclasse di una classe B, allora C<A> è sottoclasse di C

La relazione di sottoclasse non viene ereditata dai tipi generici in modo diretto. La relazione di sottoclasse è specifica per i tipi concreti e non viene estesa ai tipi generici. Quindi, non possiamo fare assunzioni sulla relazione di sottoclasse tra tipi generici basandoci sulla relazione di sottoclasse tra i loro parametri di tipo

5. Vero Falso

- Le dichiarazioni `var v = 3;` e `int v = 3;` sono equivalenti
- Le dichiarazioni `var v = 3;` e `dynamic v = 3;` sono equivalenti
- Le dichiarazioni `var v = 3;` e `object v = 3;` sono equivalenti

In sintesi, mentre `var` determina staticamente il tipo della variabile in fase di compilazione, `dynamic` consente il tipo dinamico a tempo di esecuzione

6. In un progetto in cui sia stato settato `<Nullable>enable</Nullable>`

Vero Falso

- `string s = null;` causa errore dinamico
- `string s = null;` può al più generare un warning del compilatore
- `string s = null!`; è corretto sia staticamente che dinamicamente e non genera warning

object è un tipo esplicito :)

testato su rider :)

7. Data una classe C con metodi `public void M(in int a1) { /* ... */ }` e `public void F(out int a0) {/* ... */ }` e costruttore di default

Vero Falso

- `new C().M(in 88);` è staticamente corretto
- `new C().F(out var zzz);` Console.WriteLine(zzz); causa errore statico
- `new C().F(out var zzz);` Console.WriteLine(zzz); causa errore dinamico nei casi in cui zzz non è inizializzato dalla chiamata di F()

compiler error

stampa a0

compiler error, non dinamico

8. Data `var q = new[] { 3, 5, -7, 0, 2 }.Select(i => i > 0 ? 2 * i : throw new Exception());`

Vero Falso

- la dichiarazione di q causa errore dinamico
- `q.Take(3);` causa errore dinamico
- `q.Count();` causa errore dinamico

error CS1955: Non-invocable member 'ArgumentException' cannot be used like a method

se si toglie il `throw` e si mette un metodo, funge. Altrimenti static error

9. Applicando la *dependency injection by constructor*, se una classe C ha un campo di tipo A

Vero Falso

- i costruttori di C devono avere almeno un parametro
- C deve avere un metodo `public void Inject()` usato dal DI container per inizializzare il campo automaticamente
- serve una *factory* per A, da usare nel costruttore di C

non è obbligatorio

Esercizio 3 (9 punti)

Per ciascuna delle seguenti affermazioni, indicate se è vera o falsa

1. Si considerino le seguenti classi dove `AnException` è una classe che estende `Exception`.

```
public class C {
    public int P { get; init; }
    public C(int x = 42) { P = x; }
    public IEnumerable<bool> M(int a) {
        for (int i = 0; i < a; i++) yield return true;
        if (a==P) throw new AnException();
        while (true) yield return false;
    }
}
[TestFixture]
public class CTest {
    [Test]
    public void T1() { Assert.That(new C().M(7), Is.Not.Empty); }
    [Test]
    public void T2() { Assert.That(new C(7).M(7), Is.Not.Empty); }
    [Test]
    public void T3() { Assert.That(
        new C(7).M(7), Throws.TypeOf<AnException>()); }
    [Test]
    public void T4() { Assert.That(
        () => new C(7).M(7), Throws.TypeOf<AnException>()); }
    [Test]
    public void T5() { Assert.That(
        () => new C(7).M(7).Any(), Throws.TypeOf<AnException>()); }
    [Test]
    public void T6() { Assert.That(
        () => new C(7).M(7).Take(10), Throws.TypeOf<AnException>()); }
    [Test]
    public void T7() { Assert.That(
        () => new C(7).M(7).ToArray(), Throws.TypeOf<AnException>()); }
    [Test]
    public void T8() { Assert.That(
        new C(7).M(7).ToArray(), Throws.TypeOf<AnException>()); }
    [Test]
    public void T9() {
        var source = new C() { P = 57 };
        Assert.That(source.M(57).Take(7).ToArray(),
            Is.EqualTo(new[] { true, true, true, true, true, true }));
    }
}
```

Vero Falso

- Il test T1 non compila perché manca il parametro nel costruttore
- Il test T1 ha successo
- Il test T1 fallisce perché la chiamata solleva eccezione
- Il test T2 ha successo
- Il test T3 ha successo
- Il test T4 ha successo
- Il test T5 ha successo

- Il test T6 ha successo
- Il test T7 ha successo
- Il test T8 ha successo
- Il test T9 ha successo
- Il test T9 non compila perché la property P è di sola lettura

2. Si consideri il seguente frammento di un progetto che usa l'Entity Framework

```
[Index(nameof(Login), IsUnique = true)]
public class Entity1 {
    public int Id { get; set; }
    public string Login { get; set; }
    public int MyKey { get; set; }
    public List<Entity3> Entities3 { get; set; }
}
public class Entity2 {
    public int Entity2Id { get; set; }
    public Entity1? Entity1 { get; set; }
}
public class Entity3 {
    [Key]
    public int MyKey { get; set; }
    public int Entity2Id { get; set; }
    public int Entity1Id { get; set; }
    public List<Entity1> Entities1 { get; set; }
}
public class Entity4 {
    public int Entity4Id { get; set; }
    public Entity2 Entity2 { get; set; }
}
public class MyDbContext : DbContext {
    public DbSet<Entity1> Entity1s { get; set; }
    public DbSet<Entity3> Entity3s { get; set; }
}
```

Vero Falso

- l'EF non può costruire il DB per questo frammento perché manca l'overriding di OnModelCreating → The default implementation of this method does nothing, but it can be overridden in a derived class such that the model can be further configured before it is locked down
- la classe Entity1 è un'entità, ovvero ha una tabella corrispondente sul DB
- la classe Entity2 è un'entità, ovvero ha una tabella corrispondente sul DB
- la classe Entity3 non è un'entità, perché il nome della chiave non segue le convenzioni
- la classe Entity4 è un'entità, ovvero ha una tabella corrispondente sul DB
- per rappresentare questo frammento, l'EF genera meno di 4 tabelle sul DB
- per rappresentare questo frammento, l'EF genera esattamente 4 tabelle sul DB
- per rappresentare questo frammento, l'EF genera più di 4 tabelle sul DB
- l'EF genera almeno una tabella che non rappresenta nessuna delle entità nel frammento → join tra 1 e 3 perché per le loro liste si ha una relazione (n,n)
- il seguente codice solleva eccezione per violazione di indice unique:


```
var x = new Entity1() { Login = "puffo" }; var y = new Entity1() { Login = "puffo" };
```
- nella tabella associata a Entity1 ci possono essere più righe con lo stesso valore di Login
- le property Entities1 in Entity3 e Entities3 in Entity1 rappresentano i due lati di una relazione molti-a-molti per l'EF
- le property Entities1 in Entity3 e MyKey in Entity1 rappresentano una relazione uno-a-molti per l'EF
- la property Entity1Id nella classe Entity3 è una chiave esterna su Entity1

La proprietà Entity1Id nella classe Entity3 rappresenta una chiave esterna che fa riferimento all'entità Entity1. Questa proprietà viene utilizzata per stabilire una relazione tra Entity3 e Entity1, dove Entity1Id rappresenta l'identificatore dell'istanza di Entity1 associata a un'istanza di Entity3. Quindi, possiamo dire che Entity1Id è una chiave esterna che collega le due entità.

La proprietà Entities1 in Entity3 rappresenta una relazione uno-a-molti, poiché indica che un'istanza di Entity3 può avere più istanze di Entity1 associate ad essa. Tuttavia, la proprietà MyKey in Entity1 non indica una relazione diretta con Entity3, ma semplicemente rappresenta una proprietà di Entity1.

Nome e Cognome:

Appello TAP del 16/2/2023

Scrivere nome, cognome e matricola sul foglio protocollo e sul foglio dei quiz. Avete a disposizione due ore e mezza.

Quiz (9 punti)

Per ciascuna delle seguenti affermazioni, indicate se è vera o falsa

1. Si consideri il seguente frammento di codice

```
public interface IMine {
    int B();
}

public class SUT { /*...*/
    public void TheMethod(int x){/*...*/}
    public SUT(IMine mine){/*...*/}
}

[TestFixture]
public class SUT_Test {
    [Test]
    public void TheMethodDoesNotThrow([Random(1,10,4)] int what) {
        var mine1 = new Mock<IMine>();
        mine1.Setup(x => x.B()).Returns(what);
        new SUT(mine1.Object).TheMethod(3);
    }

    [Test]
    public void BIIsUsed() {
        var mine = new Mock<IMine>();
        mine.Setup(x => x.B()).Returns(1024);
        new SUT(mine.Object).TheMethod(5);
        mine.Verify(s=>s.B(), Times.AtLeastOnce());
    }
}
```

PER USARE IL MOCK VERO E PROPRIO

ESISTE SOLO NEI MOCK

Vero Falso

- ✗ il metodo TheMethodDoesNotThrow non è staticamente corretto, perché non si può applicare l'attributo Random a un parametro
- ✗ il tipo di mine1.Object non corrisponde a quanto atteso dal costruttore di SUT, quindi il test TheMethodDoesNotThrow non è staticamente corretto
- ✗ il test TheMethodDoesNotThrow non è staticamente corretto perché un test deve contenere almeno una asserzione
- ✗ il test TheMethodDoesNotThrow è staticamente corretto ma totalmente inutile perché qualsiasi implementazione lo passa
- ✗ mine1.Object nel test TheMethodDoesNotThrow viene usata come un mock
 - ✗ mine1.Object nel test TheMethodDoesNotThrow viene usata come uno stub ma non un mock
 - ✗ mine.Object, all'interno del metodo BIIsUsed, è un mock
- ✗ mine.Object, all'interno del metodo BIIsUsed, è uno stub ma non un mock
 - ✗ la classe SUT_Test è staticamente corretta

CONTROLLA CHE
RITORNI UN
VALORE (CHECK
RESULT)

MENTRE MOCK
CONTROLLA COME
FUNZIONA
(BEHAVIOUR)

2. Sia dato il seguente codice

```

public interface ID {
    public string Name { get; }
    public string M();
}

public class D : ID {
    private static Random _random = new Random();
    private static string RandomName() {
        var names = new[] { "Pio", "Edo", "Ada", "Eva", "Ivo", "Zoe" };
        return names[_random.Next(0, names.Length)];
    }
    public string Name { get; init; }
    public string M() { return "ciao "+Name; }
    public D() { Name = RandomName(); }
}

public class C {
    public void M() { Console.WriteLine(new D().M()); }
    public void M(int n) { for (int i = 0; i < n; i++) M(); }
}

public class C1 {
    private ID _myD;
    public C1(ID d) { _myD = d; }
    public void M() { Console.WriteLine(_myD.M()); }
    public void M(int n) { for (int i = 0; i < n; i++) M(); }
}

```

Vero Falso

- ✗ la classe D non implementa correttamente l'interfaccia ID perché la property Name ha init oltre che get
- ✗ la classe D non rispetta i principi della dependency injection perché il costruttore non ha un parametro di tipo string
- ✗ la classe C1 è la corretta modifica della classe C ottenuta mediante dependency injection
- ✗ la classe C non è staticamente corretta perché non ha nessun costruttore
- ✗ le chiamate `new C().M(3);` e `new C1(new D()).M(3);` producono sempre lo stesso risultato
- ✗ le chiamate `new C().M(3);` e `new C1(new D()).M(3);` non sono mai equivalenti, perché la prima genera errore non esistendo il costruttore senza parametri per C
- ❓ la classe C1 rispetta i principi della dependency injection (senza prendere in considerazione la sua corrispondenza con la classe C)
- ✗ la classe C rispetta i principi della dependency injection
- ✗ per modificare la classe C secondo la dependency injection mantenendone il comportamento serve una factory per oggetti di tipo D

Unless the class is static, classes without constructors are given a public parameterless constructor by the C# compiler in order to enable class instantiation

è vera, ma non per il motivo della prof, ma perchè è vera quella prima!!!

La classe C non rispetta i principi della dependency injection perché crea direttamente una nuova istanza di D nel suo metodo M(), invece di accettare un'istanza di ID come parametro.

Per modificare la classe C secondo i principi della dependency injection mantenendone il comportamento, è possibile utilizzare una factory per creare gli oggetti di tipo D