Self-Study

1. **Why when creating an object from a collection and then adding an element , the size becomes 4 ?**

   In C#, when creating an object from a collection like a List<T>, the size or "Capacity" might become 4 when adding elements because of how the internal array of the List is managed. The List<T> uses an internal array to store its elements. When the number of elements added (Count) exceeds the current capacity of this internal array, the List automatically increases its capacity by reallocating a new, larger internal array (commonly doubling the existing capacity) and copying the existing elements to this new array before adding the new element.

   This means that even if you add fewer elements, the capacity might be set to a default initial size such as 4 to optimize performance and reduce the need for frequent resizing. So, the capacity grows in chunks to accommodate new elements efficiently, which can cause the size reported (capacity) to be larger than the actual number of elements (count) in the collection.

   What Particularly 4?

   The reason for picking 4 stems from design trade-offs. Raising capacity in larger chunks (instead of 1 by 1) greatly reduces the number of memory reallocations and array copying, while keeping overhead low if the list remains small. Four is considered a sweet spot that balances these factors well, hence it is hardcoded as the starting capacity in the .NET source code as the internal value `DefaultCapacity`.

2. What does dynamic keyword do ?

In C#, the dynamic keyword declares a variable whose type is resolved at runtime rather than compile time. This means the compiler skips type checking for that variable during compilation and defers it until the program is running. Therefore, a dynamic variable can hold any type, and operations on it (like method calls or property access) are bound dynamically. This provides flexibility to interact with objects when types are not known until runtime, such as when dealing with COM objects, reflection, or dynamic languages interoperability.

For example, a dynamic variable can change the type of its content during execution without compile errors:

```
dynamic value = 10;
value = "Hello";
```

The compiler allows both assignments because it checks types only when the program runs.

Using dynamic is powerful but comes with the risk of runtime exceptions if invalid operations are performed on the dynamic type because no compile-time safety checks occur.

In summary, dynamic enables late binding and bypasses static type safety, useful for

3. When creating an anonymous type, how does the CLR dispose it ? Does it require manual interaction to be disposed ?

In C#, anonymous types are compiler-generated reference types with read-only properties and no explicit name.

Regarding disposal, anonymous types do not require manual disposal nor have a special disposal mechanism because:

- They only contain managed data without unmanaged resources.

- They do not implement IDisposable.
- The CLR treats them like any regular reference type and relies on the garbage collector to reclaim their memory automatically when no longer in use.

Therefore, the CLR disposes of anonymous types via its normal garbage collection process, and no programmer interaction is needed to dispose or clean them up. They are cleaned like other standard objects once they become unreachable.

4. What is Yielding in C# ?

In C#, yield is a keyword used in an iterator method to produce a sequence of values one at a time, enabling stateful, lazy iteration over a collection.

When a method uses yield return, it returns one element at a time and preserves the current position in the iteration. Control is returned to the caller (e.g., a foreach loop) until the next element is requested, making the method resume from where it left off without creating and storing the entire collection in memory first.

There are two main forms:

- yield return <expression>;: returns an element and pauses execution until the next element is requested.
- yield break;: ends the iteration early.

This approach improves memory efficiency by generating values on demand, rather than precomputing or storing all values upfront.

Example:

```
using System;
using System.Collections.Generic;
```

```
class Program
{
    // Iterator method using yield return
    static IEnumerable<int> GetPositiveNumbers()
    {
        List<int> numbers = new List<int> { -1, -4, 3, 5 };

        foreach (var num in numbers)
        {
            if (num >= 0)
            {
                yield return num; // Return positive number and pause
                Console.WriteLine("Returned {0}", num); // Execution
resumes here on next call
            }
        }
    }

    static void Main()
    {
        foreach (var item in GetPositiveNumbers())
        {
            Console.WriteLine("Received {0}", item);
        }
    }
}
```

Explanation:

- The method `GetPositiveNumbers` uses `yield return` to return positive numbers from the list one at a time.
- When `yield return` is executed, the current value is returned to the caller (`foreach` loop).
- The method's execution pauses, and upon the next iteration, it resumes right after the `yield return`.

- The `Console.WriteLine("Returned {0}", num)` inside the iterator runs after returning each item, showing where the method resumes.

Output:

```
Received 3
Returned 3
Received 5
Returned 5
```

This example shows how `yield` creates a lazy iterator, producing values on demand without creating an intermediate collection, improving memory efficiency.

5. What is Lazy Loading in C# ?

Lazy Loading

- **Definition**: Related data is fetched only when accessed, typically via navigation properties.
- **Pros**:
    - Reduces initial query size and memory usage if related data isn't needed.
    - Simplifies initial queries.
- **Cons**:
    - Can lead to multiple database queries (N+1 problem), impacting performance.
    - Requires an active database context, or it throws exceptions if the context is disposed.
- **When to Use**: When related data is rarely accessed or to minimize initial load.

Eager Loading

- **Definition**: Related data is fetched upfront using a single query (via SQL JOINs).
- **Pros**:
    - Fewer database queries, avoiding the N+1 problem.
    - Better performance when related data is frequently accessed.
- **Cons**:
    - Higher memory usage and slower initial query if fetching unneeded data.
    - More complex queries can strain the database.
- **When to Use**: When related data is always or frequently needed.

Example Scenario

We have two models in a C# Entity Framework Core application:

- Blog: Represents a blog post.
- Comment: Represents comments on a blog post (one-to-many relationship).

Models:

```
public class Blog
{
   public int Id { get; set; }
   public string Title { get; set; }
   public string Content { get; set; }
   public List<Comment> Comments { get; set; } = new List<Comment>();
}

public class Comment
{
   public int Id { get; set; }
```

```
    public int BlogId { get; set; }
    public string Text { get; set; }
    public Blog Blog { get; set; }
}
```

DbContext:

```
using Microsoft.EntityFrameworkCore;

public class BlogContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Comment> Comments { get; set; }

    public BlogContext(DbContextOptions<BlogContext> options) :
base(options) { }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .HasMany(b => b.Comments)
            .WithOne(c => c.Blog)
            .HasForeignKey(c => c.BlogId);
    }
}
```

Database Setup:

- One blog post: Title = "My First Post".
- Two comments: Text = "Great post!" and Text = "Thanks for sharing!".

Setup for Lazy Loading:

1. Install the Microsoft.EntityFrameworkCore.Proxies package.
2. Configure the DbContext to use lazy loading proxies:

```
services.AddDbContext<BlogContext>(options =>
    options.UseSqlServer(connectionString)
           .UseLazyLoadingProxies());
```

Lazy Loading Example

In lazy loading, EF Core fetches related Comments only when the Comments navigation property is accessed.

Code:

```
using Microsoft.EntityFrameworkCore;
using System;
using System.Linq;

class Program
{
   static void Main()
   {
       var options = new DbContextOptionsBuilder<BlogContext>()
           .UseSqlServer("Your_Connection_String")
           .UseLazyLoadingProxies()
           .Options;

       using (var context = new BlogContext(options))
       {
           // Fetch a blog post
           var blog = context.Blogs.FirstOrDefault(b => b.Title == "My
First Post");

           // Access comments (triggers a new database query)
           var comments = blog.Comments;

           // Print results
           Console.WriteLine($"Blog: {blog.Title}");

           foreach (var comment in comments)
           {
               Console.WriteLine($"Comment: {comment.Text}");
           }
       }
   }
}
```

What Happens:

1. The first query fetches only the blog:
2. `SELECT TOP(1) * FROM Blogs WHERE Title = 'My First Post';`
3. Accessing blog.Comments triggers a separate query:
4. `SELECT * FROM Comments WHERE BlogId = @blogId;`

Output:

```
Blog: My First Post
Comment: Great post!
Comment: Thanks for sharing!
```

SQL Queries Executed:

- Two queries: one for the blog, one for the comments.
- For 100 blogs, accessing comments would result in 101 queries (1 for blogs + 100 for comments, N+1 problem).

Eager Loading Example

In eager loading, we use .Include() to fetch related Comments in the same query as the Blog.

Code:

```
using Microsoft.EntityFrameworkCore;
using System;
using System.Linq;

class Program
{
   static void Main()
   {
       var options = new DbContextOptionsBuilder<BlogContext>()
```

```
            .UseSqlServer("Your_Connection_String")
            .Options;

        using (var context = new BlogContext(options))
        {
            // Fetch a blog post with its comments eagerly
            var blog = context.Blogs
                .Include(b => b.Comments)
                .FirstOrDefault(b => b.Title == "My First Post");

            // Access comments (no additional query needed)
            var comments = blog.Comments;

            // Print results
            Console.WriteLine($"Blog: {blog.Title}");

            foreach (var comment in comments)
            {
                Console.WriteLine($"Comment: {comment.Text}");
            }
        }
    }
}
```

What Happens:

1. A single query fetches the blog and its comments using a JOIN:

```
SELECT TOP(1) b.*, c.*
FROM Blogs b
LEFT JOIN Comments c ON b.Id = c.BlogId
WHERE b.Title = 'My First Post';
```

2. No additional queries are made when accessing blog.Comments.

Output:

```
Blog: My First Post
Comment: Great post!
Comment: Thanks for sharing!
```

SQL Queries Executed:

- One query combining blogs and comments.
- For 100 blogs, you'd still get one query, avoiding the N+1 problem.

Additional Notes

- **N+1 Problem**: Lazy loading can cause performance issues in loops. For example, fetching comments for 100 blogs results in 101 queries. Eager loading with .Include() avoids this.
- **EF Core Specifics**:
    - Lazy loading requires Microsoft.EntityFrameworkCore.Proxies and UseLazyLoadingProxies().
    - Eager loading uses .Include() for direct relationships and .ThenInclude() for nested relationships.
    - For complex queries, Explicit Loading (using .Load()) is an alternative, loading related data on demand but explicitly.

Choosing Between Them:

- Use **lazy loading** when related data is infrequently accessed or to reduce initial query overhead.
- Use **eager loading** when related data is always needed or to optimize performance in loops.