Beyond the basic Singleton, Scoped, and Transient lifetimes in .NET Core's Dependency Injection (DI), there are advanced concepts and techniques that give you more control over how dependencies are managed. Here are some important ones:

1. Factory-Based Dependency Injection

Instead of just registering a service with a defined lifetime, you can use factory methods to customize object creation.

Example:

```
services.AddSingleton<IMyService>(provider =>
{
    var config = provider.GetRequiredService<IConfiguration>();
    return new MyService(config["SomeSetting"]);
});
```

2. Conditional Registration (Based on Environment or Other Factors)

You might want to register different implementations based on configuration settings.

Example:

```
if (Environment.GetEnvironmentVariable("ASPNETCORE_ENVIRONMENT") == "Development")
{
    services.AddSingleton<INotificationService, EmailNotificationService>();
}
else
{
    services.AddSingleton<INotificationService, SmsNotificationService>();
}
```

3. Named or Keyed Services (Using IServiceProvider)

Since built-in DI in .NET Core doesn't directly support named dependencies (unlike Autofac), you can use IServiceProvider to manually resolve dependencies.

Example:

```
services.AddTransient<INotificationService, EmailNotificationService>();
services.AddTransient<INotificationService, SmsNotificationService>();

services.AddTransient<Func<string, INotificationService>>(serviceProvider => key =>
{
    return key switch
    {
        "Email" => serviceProvider.GetRequiredService<EmailNotificationService>(),
        "SMS" => serviceProvider.GetRequiredService<SmsNotificationService>(),
        _ => throw new ArgumentException("Invalid service type")
    };
});
```

4. Open-Generic Registrations

If you want to register a generic type but resolve it with different type parameters, you can use open-generics.

Example:

```
services.AddSingleton(typeof(IRepository<>), typeof(GenericRepository<>));
```

5. Service Decorators (Using IServiceCollection.Decorate in Scrutor)

.NET Core's DI doesn't natively support decorators, but you can achieve it using Scrutor.

Example (Using Scrutor NuGet Package):

```
services.AddTransient<IService, ServiceImplementation>();
services.Decorate<IService, LoggingServiceDecorator>();
```

6. Resolving Services Manually (Using IServiceProvider)

Although constructor injection is preferred, you can use IServiceProvider for dynamic resolution.

Example:

```
var serviceProvider = services.BuildServiceProvider();
using (var scope = serviceProvider.CreateScope())
{
    var myService = scope.ServiceProvider.GetRequiredService<IMyService>();
    myService.DoSomething();
}
```

7. Custom Lifetime Management

You can control when instances are disposed using IHostApplicationLifetime or by implementing IDisposable.

Example:

```
public class MyService : IDisposable
{
    public void Dispose()
    {
        // Clean up resources
    }
}
```

8. Registering Multiple Implementations of the Same Interface

.NET Core's DI allows you to register multiple implementations and resolve them as an IEnumerable<T>.

Example:

```
services.AddTransient<INotificationService, EmailNotificationService>();
services.AddTransient<INotificationService, SmsNotificationService>();
```

Resolving:

```
var services = serviceProvider.GetRequiredService<IEnumerable<INotificationService>>();
foreach (var service in services)
{
    service.Notify("Hello");
}
```

Conclusion:

Beyond Singleton, Scoped, and Transient, you can enhance .NET Core DI with factories, conditional services, named dependencies, open-generics, decorators, manual resolution, custom lifetimes, and multi-implementation registrations. These techniques allow for more flexible and scalable dependency injection in enterprise applications.