1. Extensions methods

Extension methods allow you to "extend" existing types with new methods, without modifying the type itself, creating a subclass, or recompiling the type. This is particularly useful when you want to add methods to types that you do not own, such as classes in .NET Framework or any third-party library.

To create an extension method:

- You define a static method in a static class.
- The first parameter of the method specifies the type the method operates on, prefixed with the `this` keyword.

```
public static class StringExtensions
{
    public static int WordCount(this string str)
    {
        return str.Split(new char[] { ' ', '.', '?' },
StringSplitOptions.RemoveEmptyEntries).Length;
    }
}
```

Usage:

```
string example = "Hello, world!";
int count = example.WordCount();  // Outputs 2
```

In the example above, `WordCount` method is an extension method added to the `string` type, allowing any string instance to call it as if it were an instance method.

2. Anonymous types

Anonymous types provide a convenient way to encapsulate a set of read-only properties into a single object without having to explicitly define a type first.

They are typically used in select queries of a LINQ statement where the result is a subset of the properties of the objects being queried.

You create an anonymous type with the `new` keyword followed by an object initializer specifying properties and their values.

```
var book = new { Title = "1984", Author = "George Orwell" };
Console.WriteLine($"Book: {book.Title}, Author:
{book.Author}");
```

Here, `book` is an object of an anonymous type with `Title` and `Author` properties. The type of `book` is determined by the compiler and is read-only, meaning after instantiation, these properties cannot be changed.

3. Lambda expressions

Lambda expressions are a concise way to represent an anonymous method. These expressions are especially useful for writing LINQ query expressions and to define delegates or expression tree types.

The basic syntax of a lambda expression includes parameters on the left side of the lambda operator `=>`, and an expression or a statement block on the right.

```
Func<int, int, int> add = (x, y) => x + y;
Console.WriteLine(add(5, 3));  // Outputs 8
```

In this example, `add` is a delegate that takes two integers and returns their sum. The lambda expression `(x, y) => x + y` defines how this delegate adds its two parameters.

4. Language Integrated Query (LINQ)

LINQ, or Language Integrated Query, is a set of technologies introduced in .NET Framework 3.5 that adds native data querying capabilities to .NET languages, especially C#. It allows developers to write queries directly within the C# language.

By default LINQ allows us to write queries against any data source which implement the IEnumerable interface.

4.2. Working with other data sources

On top of LINQ there are many additional libraries which allows us to query other data sources like relational database, XML file etc.

4.3. History

LINQ was first released in November 2007 as a part of .NET Framework 3.5. The idea behind LINQ was to bring different types of data querying into a single, unified, strongly typed programming model. Before LINQ, handling data typically required switching context depending on the data source (e.g., SQL for databases, XPath for XML), and integrating the queried data back into the programming language in use. LINQ was designed to streamline this process by integrating query functionality directly into C#, making the code cleaner and reducing errors from data type mismatches.

4.4. How LINQ works?

LINQ introduces patterns for querying and updating data. It can be applied to various data sources by implementing its standard query operators, which operate on sequences of data. The sequences are objects that implement the `IEnumerable<T>` or `IQueryable<T>` interfaces. LINQ providers exist for several types of data sources, including:

- Objects in memory (LINQ to Objects)
- Relational databases (LINQ to SQL, LINQ to Entities)
- XML documents (LINQ to XML)
- Datasets (LINQ to DataSet)

4.5. LINQ syntaxes

LINQ provides two main syntaxes for writing queries: Query Syntax and Method Syntax.

4.5.1. Query syntax (Comprehension Syntax)

Query syntax is similar to SQL and is sometimes more readable, especially for those familiar with SQL. It uses query operators like `from`, `select`, `where`, `join`, `group`, and `orderby`.

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5, 6 };
var evenNumbers = from num in numbers
                  where num % 2 == 0
                  select num;
```

This example finds all even numbers in a list.

4.5.2. Method syntax (Extension method syntax)

Method syntax uses extension methods provided by the `System.Linq` namespace. These methods include `Where`, `Select`, `OrderBy`, `Join`, `GroupBy`, etc., and can be chained together to write complex queries.

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5, 6 };
var evenNumbers = numbers.Where(num => num % 2 == 0).ToList();
```

This example, like the one above, finds all even numbers in a list but uses method syntax.

4.6. Choosing Between Syntaxes

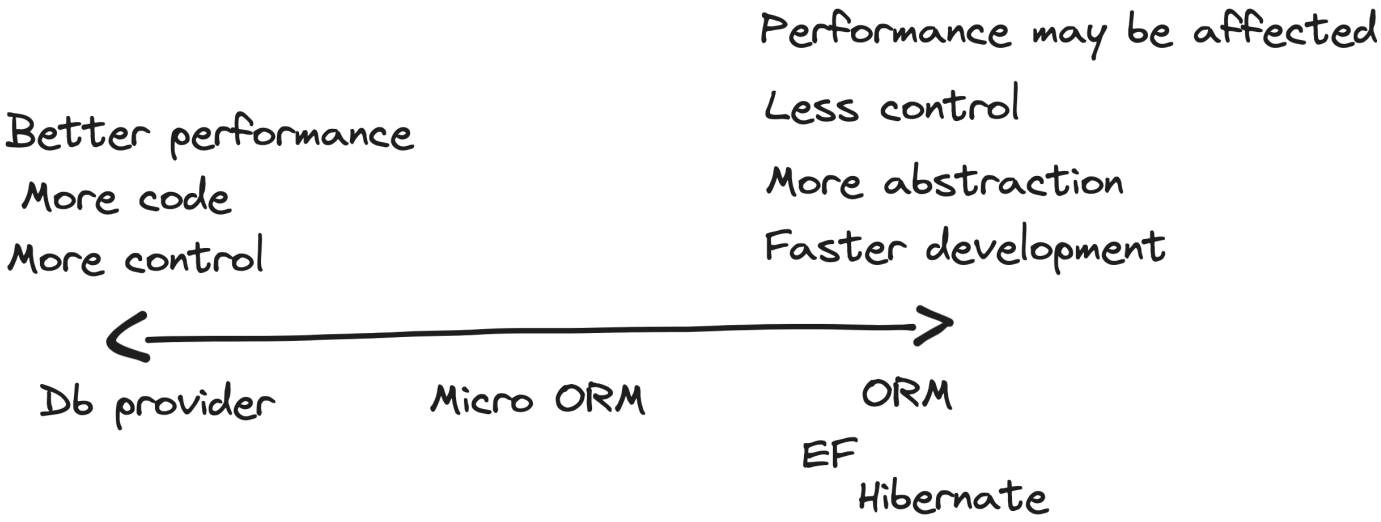
Both syntaxes can be used to accomplish the same tasks, but the choice between them often depends on personal preference or the specific scenario:

- **Query Syntax** is typically more concise for complex queries, such as those involving joins or grouping.
- **Method Syntax** provides more flexibility and can be easier to use for simple queries or when dynamically building queries.

4.7. More LINQ examples

You can find more LINQ examples under the link below.
https://linqsamples.com/

5. Object-relational mapping



ORM stands for Object-Relational Mapping, a programming technique used to convert data between incompatible type systems in object-oriented programming languages. In simple terms, an ORM is a tool that allows you to query and manipulate data from a database using an object-oriented paradigm. When you use an ORM, you work with objects in your programming language, rather than dealing directly with database tables and SQL queries.

ORMs help developers to abstract the complexities involved in performing CRUD (Create, Read, Update, Delete) operations in the database, enabling them to focus more on business logic rather than data management. They typically provide mechanisms to:

- Map application classes to database tables.
- Automatically generate SQL queries.
- Handle transactions.
- Manage object state.
- Facilitate relationships between objects (e.g., one-to-many, many-to-many).

Impedance mismatch*
The impedance mismatch problem refers to the difficulties that arise when trying to integrate systems that use different data representations or architectural principles. Originally a term from electrical engineering, it has been co-opted into the field of computer science, particularly in the context of object-relational mapping (ORM).

5.1. Advantages of Using ORM

- **Productivity**: Automatically handling database interactions and CRUD operations saves development time.
- **Maintainability**: Code is generally cleaner, more modular, and easier to update.
- **Portability**: ORMs abstract most of the database interactions, making the application more portable across various database systems.

5.2. Disadvantages of Using ORM

- **Performance**: ORMs might generate less optimized SQL queries compared to hand-tuned SQL, particularly for complex queries.
- **Complexity**: Complex relational data structures can be difficult to manage effectively with some ORMs.
- **Control**: Developers might find themselves limited by the ORM's design and abstraction, making specific optimizations or queries hard to implement.

5.3. Micro ORM

Micro ORMs are a subset of ORM tools that provide a lighter weight, less abstracted approach to ORM. They typically offer some of the convenience of full-feature ORMs but with more direct control over SQL and fewer automations. This can lead to better performance and more flexibility while still sparing developers from writing repetitive SQL code.

5.3.1. Examples of Micro ORMs:

- **Dapper**: A simple object mapper for .NET that extends the IDbConnection interface. It provides virtually no DB schema configuration or change tracking capabilities, focusing instead on performance and simplicity.
- **PetaPoco**: A tiny ORM for .NET that mostly targets single table operations but supports multi-PoCo queries and mappings.
- **Sql2o**: A small Java library for executing SQL queries and mapping the result to Java objects, focusing on simplicity and performance.

5.3.2. Advantages of Micro ORMs

- **Performance**: They generally produce more efficient SQL because they involve less automatic query generation and more direct developer control.
- **Simplicity**: They are easier to understand and use effectively when compared to full-feature ORMs, especially for developers familiar with SQL.
- **Flexibility**: They allow more direct control over SQL, making it easier to optimize specific queries or use database-specific features.

5.3.3. Disadvantages of Micro ORMs

- **More SQL Code**: While they reduce some boilerplate, they require more hand-written SQL than traditional ORMs.
- **Fewer Features**: They typically do not support advanced ORM features like change tracking, lazy loading, or automatic relationship handling.

6. Entity Framework

Entity Framework (EF) is a popular Object-Relational Mapping (ORM) framework for .NET developers. It serves as an abstraction that allows developers to work with data in the form of domain-specific objects and properties, without having to deal with the underlying database tables and SQL queries directly. This can significantly simplify data access within software applications and reduce the amount of boilerplate code that developers need to write.

5.1. Advantages and disadvantages of Entity Framework

5.1.1. Advantages of Entity Framework

1. **Increased Productivity**: EF reduces the amount of boilerplate code developers need to write for data access operations. With features like automatic migrations, LINQ integration, and a powerful scaffolding system, developers can accomplish more with less code.

2. **Data Model Consistency**: EF ensures consistency between the application data model and the underlying database through its data annotations and fluent API configurations. This consistency helps maintain data integrity and reduces errors.
3. **Abstraction from SQL**: EF abstracts the database operations into .NET code, meaning developers can work with a consistent programming model without deep knowledge of SQL. This abstraction is particularly beneficial for teams with strong .NET skills but less experience in database management.
4. **Strongly Typed Operations**: Thanks to integration with LINQ, EF allows querying databases using strongly typed .NET code. This approach provides compile-time checking of queries, reducing runtime errors and improving code quality.
5. **Support for Complex Mappings and Relationships**: EF can automatically handle complex mappings between your database and your objects, manage relationships, and propagate updates/inserts/deletes through your object graph.
6. **Support for Multiple Database Backends**: EF supports multiple database systems, which allows for easier switching between different databases if needed and can aid in targeting various environments (like SQL Server for development and PostgreSQL for production).

5.1.2. Disadvantages of Entity Framework

1. **Performance Overheads**: The abstraction layer can sometimes lead to suboptimal SQL queries that are not as efficient as hand-written SQL, particularly for complex queries or large batch operations. This can be a significant drawback in high-load scenarios.
2. **Steep Learning Curve**: While basic operations are straightforward, mastering EF's advanced features (like complex mappings, custom migrations, and performance tuning) requires a significant investment in learning.
3. **Less Control Over the Database Operations**: EF's abstraction can be a double-edged sword. It simplifies many operations but can obscure

what happens at the database level, making it harder to optimize or debug performance issues.

4. **Migration Challenges**: Although EF migrations are powerful, they can sometimes generate unexpected or inefficient schema changes that require manual adjustment. This can be especially problematic in large, complex databases with high availability requirements.
5. **Memory Consumption**: EF can consume more memory due to its tracking of changes in objects. This might not be ideal for scenarios with large data sets or where minimal resource consumption is critical.
6. **Potential for Vendor Lock-in**: While EF supports multiple databases, switching between providers can sometimes expose inconsistencies because not all SQL features are supported identically across providers.

5.2. Code first vs database first approach

When using ORM we often have two ways of working with the database.

5.2.1. Code First Approach

Code First is a development approach where you first create and define your entity classes and relationships in your application code using C# (or another .NET language). The database schema is then generated from these classes using migrations.
In other words the structure of the database is directly generated from code.

How it works:

1. **Define Models**: You define your domain models as POCO (Plain Old CLR Objects) classes in your application. These classes include properties that map to the columns of the database table.
2. **Define Context**: You create a `DbContext` class where you define `DbSet` properties for each entity. This class manages the database operations.
3. **Generate Database**: Using EF migrations, you generate the database schema directly from your models. Migrations allow you to evolve your database schema as your models change over time without losing data.

Advantages:

- **Agility**: It is highly agile as changes to the database are made programmatically through the code.
- **Version Control**: Changes to the database schema are version-controlled as migrations are just code.
- **Development Speed**: Faster development and easier to manage in a development team as the focus is purely on the code.

Disadvantages:

- **Control Over Database Design**: You might have less control over the database design as it's a by-product of your domain classes.
- **Initial Learning Curve**: Requires understanding of EF migrations and how to manipulate database operations through code.

5.2.2. Database First Approach

Database First is a methodology where the database schema is created independently of the application code, often using a database management tool or SQL directly. EF then generates the entity classes and a `DbContext` that map to the existing database schema.
In other words database design is created separately by database administrator.

How it works:

1. **Create Database Schema**: You directly create tables, relationships, constraints, and other database elements using a database design tool or through SQL scripts.
2. **Generate Models**: Using EF tools, you generate the entity classes and a `DbContext` that match your existing database schema.
3. **Synchronize Changes**: When changes are made to the database schema, you need to regenerate or manually adjust the entity models to keep them in sync with the database.

Advantages:

- **Control Over Database Design**: Gives you full control over the database design, useful in situations where database performance and

optimizations are critical.
- **Separation of Concerns**: The database can be managed by a database administrator or team focused on DB management, while developers focus on the application code.
- **Familiarity**: Suitable for teams familiar with traditional database development approaches.

Disadvantages:

- **Synchronization Issues**: Keeping the code synchronized with the database can become cumbersome, especially with frequent schema changes.
- **Less Agile**: Changes to the database require regenerating the code layer, which can slow down the development process.
- **Version Control**: Managing changes through SQL scripts can be less straightforward than code-based migrations.

5.2.1. When to use database first vs code first approach?

When database first might make more sense:

1. **Existing Database**: If your project needs to work with an existing database where the schema is already defined and contains important data, Database First is often the best choice. It allows you to generate entity models directly from the existing schema without needing to redesign or migrate data.
2. **Complex Database Designs**: For applications where the database design involves complex relationships, optimizations, or legacy stored procedures that must be retained, Database First can provide more control and precision in handling these complexities.
3. **Database-Driven Projects**: In scenarios where the database schema is likely to be managed by a specialized team of database administrators (DBAs), the Database First approach allows for clear separation of duties. DBAs can focus on optimizing the database independently from the application development.

4. **Frequent Schema Changes by DBAs**: If changes to the database schema are frequently made directly by DBAs without coordination with the development team, Database First can simplify the process of updating the application to reflect these changes.
5. **Large Teams with Separated Responsibilities**: For larger teams where responsibilities are divided clearly between developers and database specialists, Database First can help ensure that any database changes are deliberate and managed by those with the most knowledge in that area.

When database first might make more sense:

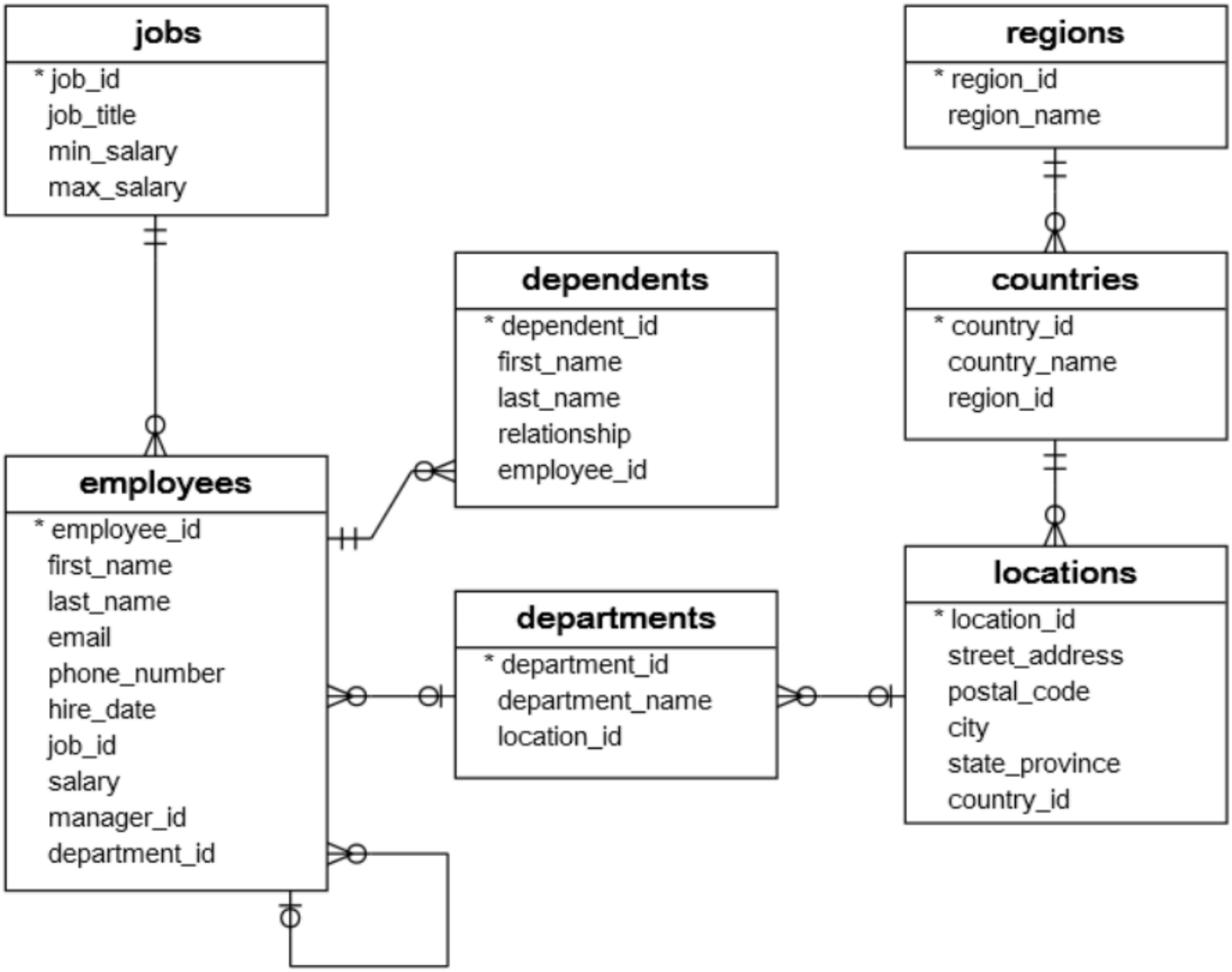
1. **New Projects**: For new projects starting from scratch, Code First allows developers to work from the ground up without needing an existing database schema. It can be more agile and flexible, allowing for rapid iteration on the development of the application and its data model.
2. **Projects Requiring Agile Development**: If your project needs to adapt quickly to changing requirements, Code First offers a more agile approach. Developers can modify the entity models and quickly propagate changes to the database using migrations, facilitating rapid development cycles.
3. **Version Control and Collaboration**: Code First supports version-controlled migrations, making it easier to track and collaborate on changes to the database schema across development teams. This can be particularly beneficial in environments where continuous integration and deployment are practiced.
4. **Simplicity and Reduced Database Complexity**: When the application does not require a highly optimized database schema or when database performance is not the primary concern, Code First can simplify development by abstracting much of the database interaction.
5. **Education and Prototyping**: For educational purposes or rapid prototyping, Code First is excellent as it allows for quick setup and iteration without needing deep database knowledge.

Best Practices and Considerations

- **Hybrid Approaches**: Some projects might benefit from a hybrid approach, where the database is initially designed and created using Database First, and then managed moving forward with Code First. This can be effective when transitioning from an existing schema to a more agile development process.
- **Performance Considerations**: Both approaches require an understanding of how EF translates operations to SQL. Especially with Code First, developers need to be aware of the potential for generating inefficient queries and plan accordingly.
- **Learning Curve**: The learning curve for Code First might be steeper for those unfamiliar with ORM concepts, whereas Database First could be more intuitive for those with a strong SQL background but less familiarity with ORM patterns.

7. Example of working with the database first approach

Let's assume that we already have a database setup with all the tables setup by the database administrator.



7.1. Preparing the project

1. Install the following packages:
   Microsoft.EntityFrameworkCore
   Microsoft.EntityFrameworkCore.SqlServer
   Microsoft.EntityFrameworkCore.Design
   Microsoft.AspNetCore.Mvc.NewtonsoftJson
2. Run the following commands in terminal:

```
dotnet new tool-manifest
dotnet tool install dotnet-ef --version 8.0.0
```

3. Add the following line to ItemGroup element in csproj

```
<InvariantGlobalization>true</InvariantGlobalization>
```

4. Add connection strings to app settings.
5. Create directory Models and Context
6. Run the following command in the terminal:

```
dotnet ef dbcontext scaffold "Data Source=localhost;Initial
Catalog=APBD;User ID=sa;Password=asd123POKo223;Encrypt=False"
Microsoft.EntityFrameworkCore.SqlServer --output-dir Models --
context-dir Context
```