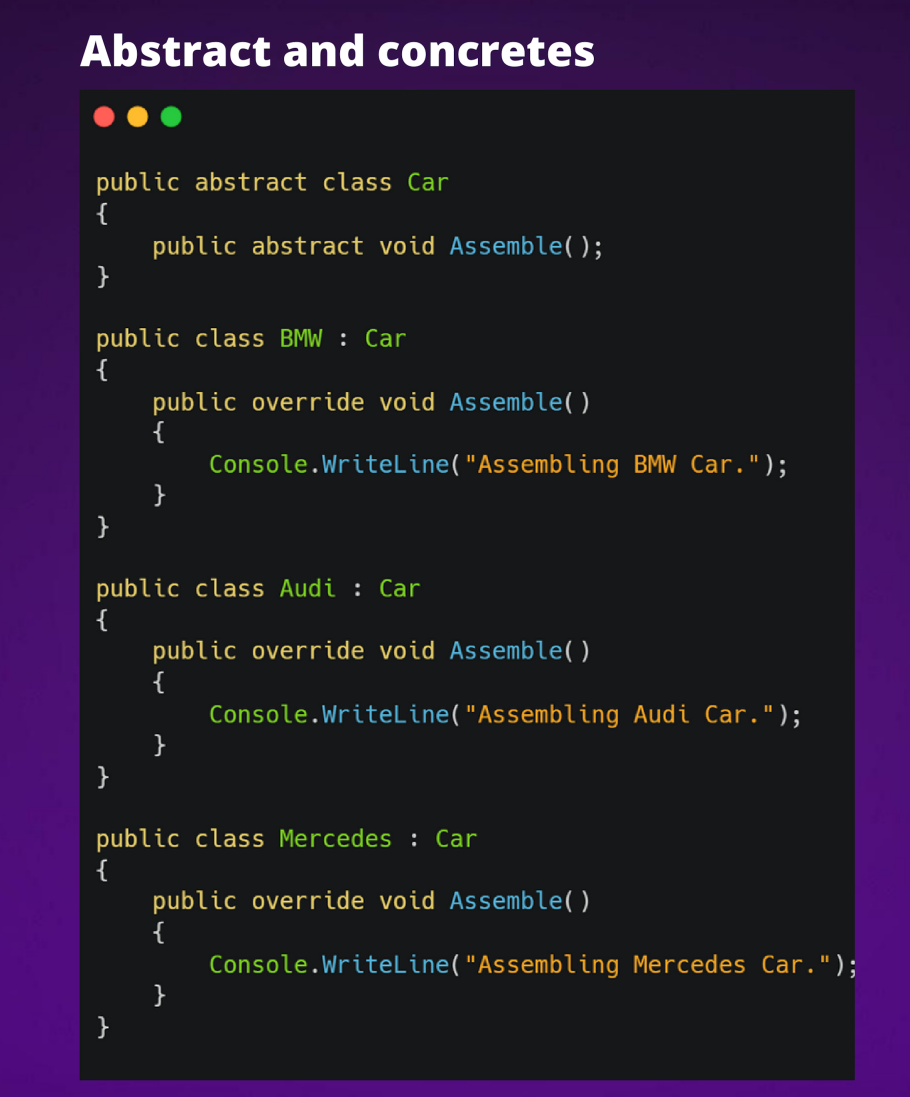
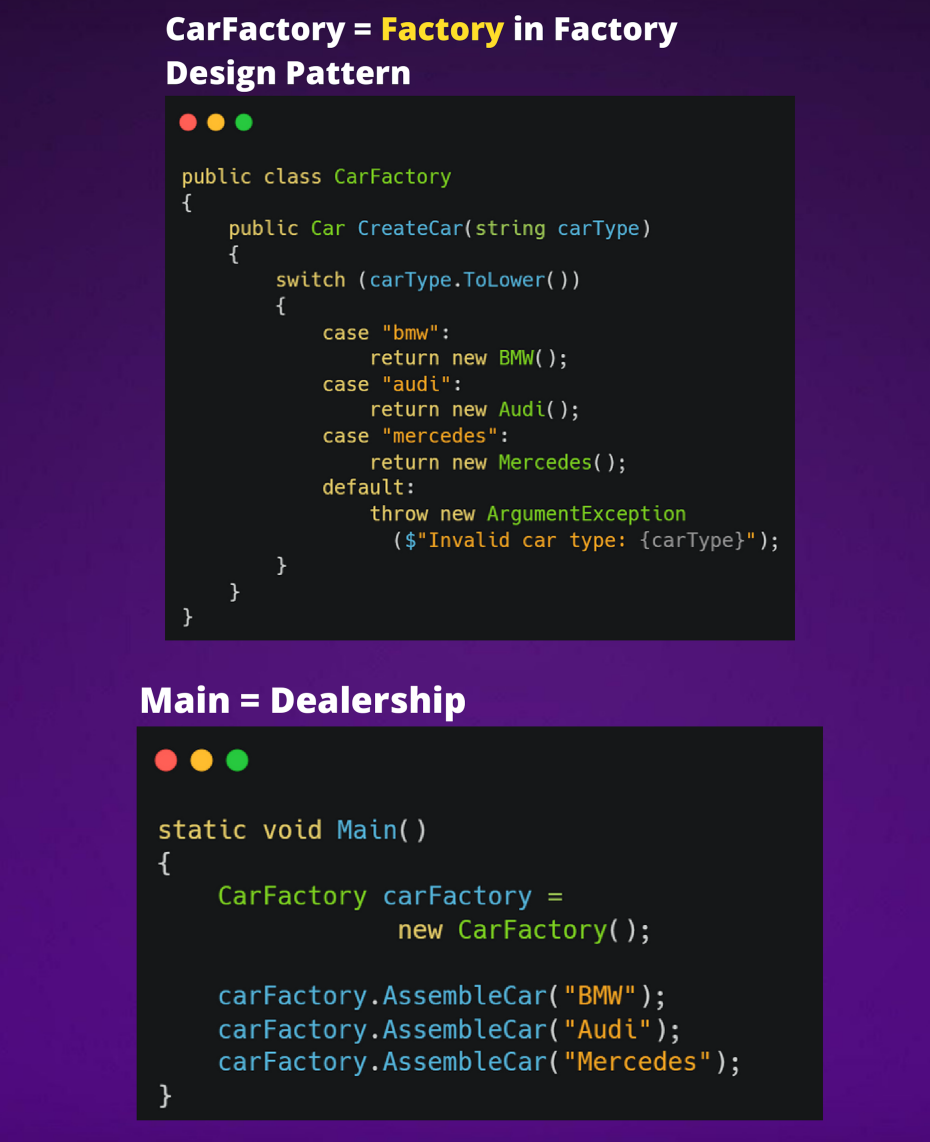
𝐏𝐚𝐭𝐭𝐞𝐫𝐧𝐬:  
  
• Factory: <https://lnkd.in/dEi4T_4Y>  
• Facade: <https://lnkd.in/dTJbask7>  
• Chain Responsibility: <https://lnkd.in/d3i2qr3Y>  
• Circuit Breaker: <https://lnkd.in/dpzAdjYH>  
• Adapter: <https://lnkd.in/dFvCddGG>  
• CQRS: <https://lnkd.in/dfdfxnCW>  
• Repository: <https://lnkd.in/dkMtAjXy>  
  
𝐐𝐮𝐢𝐜𝐤 𝐂# 𝐓𝐢𝐩𝐬:  
  
• Throw exception: <https://lnkd.in/dXw8mWhf>  
• File•Scoped Namespace: <https://lnkd.in/dPryqt6q>  
• Conditionals: <https://lnkd.in/d84vSsRu>  
• Don't repeat yourself: <https://lnkd.in/dGJGxtsH>  
• Tall not Wide: <https://lnkd.in/duveNqGY>  
• Constant Pattern Matching: <https://lnkd.in/dGg2jeHp>  
• Don't return Null Collection: <https://lnkd.in/dEHwup7x>  
• Sealed Class: <https://lnkd.in/dWVe9jKP>  
• Ref vs Out: <https://lnkd.in/dz7eVaUq>  
• Required keyword: <https://lnkd.in/dpbZbTfu>  
  
𝐆𝐞𝐧𝐞𝐫𝐚𝐥 𝐓𝐢𝐩𝐬:  
  
• API Versioning: <https://lnkd.in/dcra7XcU>  
• Http Client: <https://lnkd.in/ddxRRd8y>  
• Records: <https://lnkd.in/djgVWwDR>  
• Webhooks: <https://lnkd.in/dVpSTt8q>  
• Dependency Injection: <https://lnkd.in/dwa3b_6B>  
• Use Expression Func: <https://lnkd.in/dPe6kSQ7>  
• LINQ let: <https://lnkd.in/d4P98yJy>  
• LINQ MaxBy: <https://lnkd.in/dF7KGHKG>  
• Micro and nanoseconds: <https://lnkd.in/dCvyqzke>  
• "Turkish Problem": <https://lnkd.in/d5kAeRxm>  
  
𝐄𝐧𝐭𝐢𝐭𝐲 𝐅𝐫𝐚𝐦𝐞𝐰𝐨𝐫𝐤:  
  
• Compiled Query: <https://lnkd.in/dAAwBgjk>  
• Global Query Filter: <https://lnkd.in/dJ2p5FNv>  
• Get the fields you need: <https://lnkd.in/dyR66p_v>  
• Performance: <https://lnkd.in/dV86mZh5>  
  
𝐌𝐢𝐝𝐝𝐥𝐞𝐰𝐚𝐫𝐞:  
  
• IOptions: <https://lnkd.in/d8nfrJKi>  
• IOptionsSnapshots: <https://lnkd.in/dQHmFXQ7>  
• CORS: <https://lnkd.in/dDuZ7-_8>  
• Middleware order: <https://lnkd.in/dgpBKC_F>  
• Singleton/Scoped/Transient: <https://lnkd.in/dsC-jfZS>  
• Global Error Handling: <https://lnkd.in/dfFDnFdm>  
• ASP .NET Pipeline: <https://lnkd.in/dGrRPwu4>  
• RateLimiter: <https://lnkd.in/d4xDpjND>  
  
𝐏𝐞𝐫𝐟𝐨𝐫𝐦𝐚𝐧𝐜𝐞:  
  
• Cancellation Token: <https://lnkd.in/dCC6yvNQ>  
• Immutable Collections: <https://lnkd.in/dC-xaK_Q>  
• HttpRequest Body: <https://lnkd.in/dsCysex2>  
• Task.WhenAll(): <https://lnkd.in/dN6SzXUX>  
• StringBuilder: <https://lnkd.in/dERnQAAQ>  
• gRPC: <https://lnkd.in/dth5MuDu>  
  
𝐏𝐚𝐜𝐤𝐚𝐠𝐞𝐬:  
  
• Hangfire: <https://lnkd.in/dcaJUpq2>  
• MediatR Notifications: <https://lnkd.in/dXfkubUQ>  
• FluentEmail: <https://lnkd.in/dzMdZeb8>  
• Bogus: <https://lnkd.in/dn5YbVRJ>

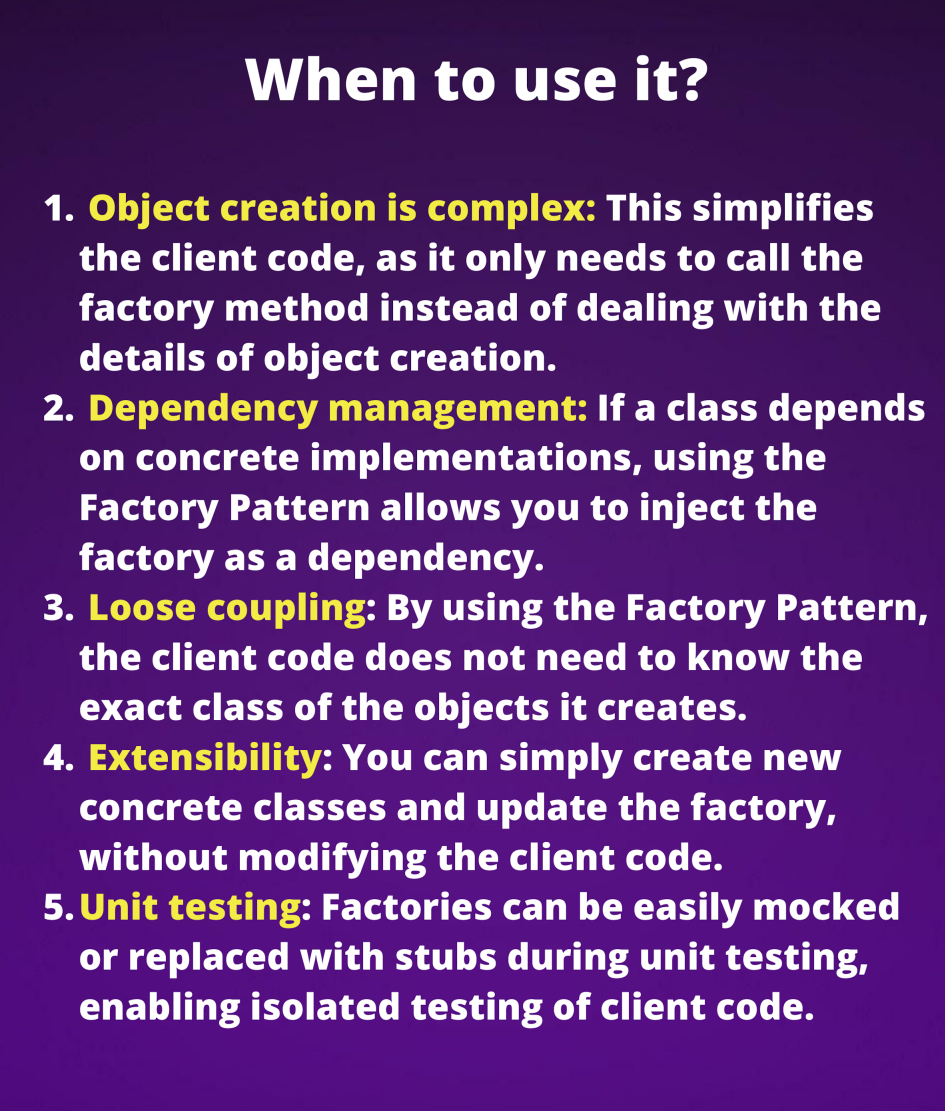
1. **What is the Factory Design pattern?**  
     
   Explained. Simplified.  
     
   The Factory Design Pattern is a creational design pattern that provides an interface for creating objects without specifying their concrete classes. It encapsulates the object creation logic in a separate class (the factory) and allows the client code to create objects through this factory interface. This pattern promotes loose coupling and enhances code flexibility by decoupling the client code from the actual object creation process.  
     
   Key components of the Factory Design Pattern:  
     
   1. Product (Car in our example): The interface or abstract class that represents the objects being created by the factory.  
   2. Concrete Products (BMW, Audi, Mercedes in our example): The specific implementations of the product interface.  
   3. Factory (CarFactory in our example): The class responsible for creating objects (concrete products) that implement the product interface. It abstracts the object creation process from the client code.



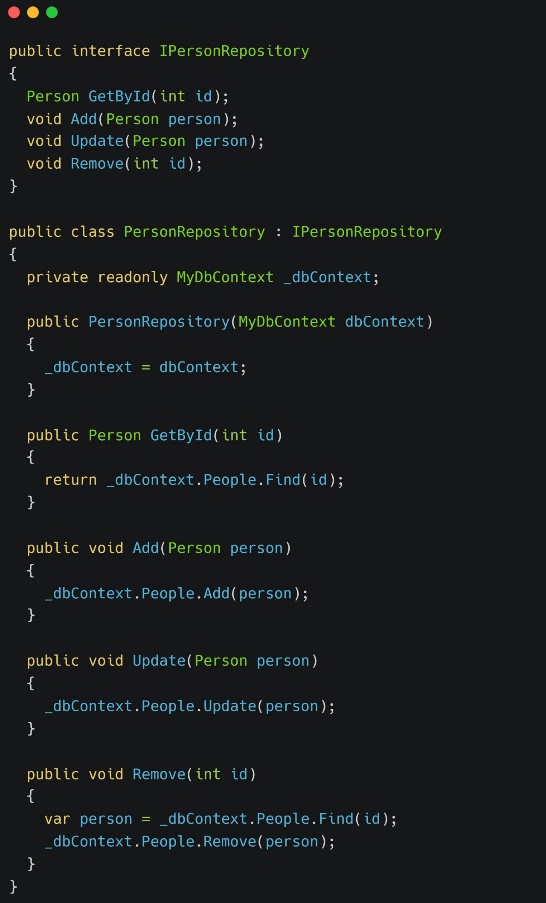






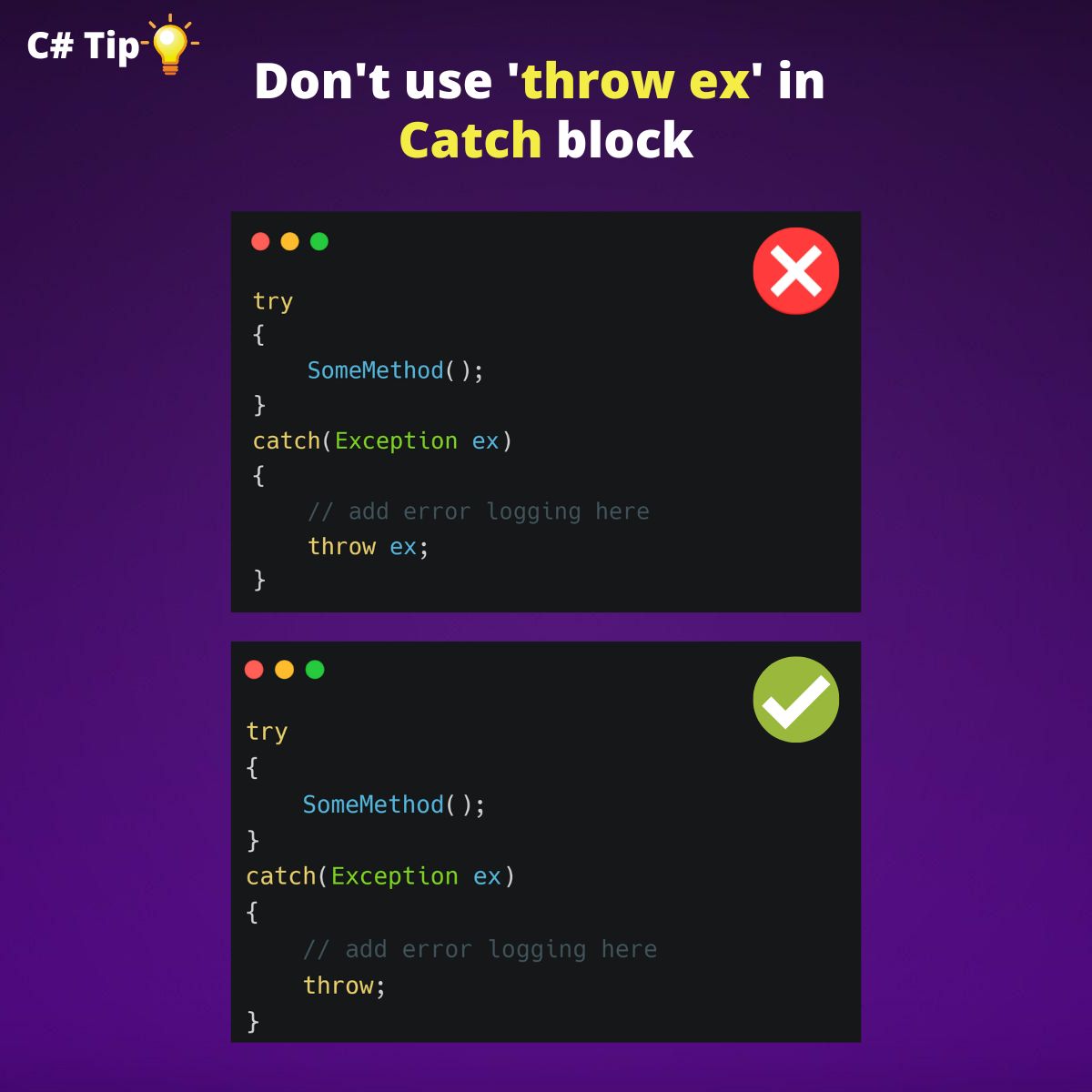


1. **Facade Pattern - 𝐒𝐢𝐦𝐩𝐥𝐢𝐟𝐢𝐞𝐝**  
     
   The Facade pattern is a structural pattern that provides a simplified interface to a complex system of classes, subsystems, or APIs.  
     
   The pattern encapsulates the complexity of the system behind a single class, called the facade, which acts as an entry point for clients to access the system's functionality.  
     
   𝐄𝐱𝐚𝐦𝐩𝐥𝐞:  
   Imagine a large corporation with multiple departments, each responsible for a specific aspect of the business.   
   There might be a sales department, a marketing department, an IT department, and a finance department.  
     
   When you have a question, complaint, or request, 𝐲𝐨𝐮 𝐜𝐚𝐥𝐥 𝐭𝐡𝐞 𝐜𝐮𝐬𝐭𝐨𝐦𝐞𝐫 𝐬𝐞𝐫𝐯𝐢𝐜𝐞 𝐝𝐞𝐩𝐚𝐫𝐭𝐦𝐞𝐧𝐭.   
   What customer service department will do?  
     
   Answer: 𝐅𝐨𝐫𝐰𝐚𝐫𝐝 𝐢𝐭 𝐭𝐨 𝐭𝐡𝐞 𝐚𝐩𝐩𝐫𝐨𝐩𝐫𝐢𝐚𝐭𝐞 𝐝𝐞𝐩𝐚𝐫𝐭𝐦𝐞𝐧𝐭.  
     
   • If you have a sales issue, the customer service representative forwards your issue to the sales department.   
   • The sales department handles your issue and then sends a response back to the customer service representative.   
   • The customer service representative then communicates the response back to you.  
     
   Similarly, the Facade pattern 𝐩𝐫𝐨𝐯𝐢𝐝𝐞𝐬 𝐚 𝐬𝐢𝐦𝐩𝐥𝐢𝐟𝐢𝐞𝐝 𝐢𝐧𝐭𝐞𝐫𝐟𝐚𝐜𝐞 𝐟𝐨𝐫 𝐜𝐥𝐢𝐞𝐧𝐭𝐬 𝐭𝐨 𝐢𝐧𝐭𝐞𝐫𝐚𝐜𝐭 𝐰𝐢𝐭𝐡 𝐚 𝐜𝐨𝐦𝐩𝐥𝐞𝐱 𝐬𝐲𝐬𝐭𝐞𝐦. The facade shields the clients from the complexity of the system's internals and provides a simplified view of the system's functionality.  
     
   Do you know what SOLID Principle's problem Facade can solve? 👇
2. **What is Chain Responsibility Pattern?**  
     
   When should I implement it?  
   How to implement it?  
     
   The Chain of Responsibility Pattern allows us to easily separate dependent parts to make code more extensible and testable.  
     
   In simple words, we can say that the chain of responsibility design pattern creates a chain of receiver objects for a given request.  
     
   In this design pattern, normally each receiver contains a reference to another receiver. If one receiver cannot handle the request then it passes the same request to the next receiver and so on. One receiver handles the request in the chain or one or more receivers handle the request.
3. **How to make the system more resilient and fault-tolerant?  
     
   Did you hear about 𝐂𝐢𝐫𝐜𝐮𝐢𝐭 𝐁𝐫𝐞𝐚𝐤𝐞𝐫 𝐏𝐚𝐭𝐭𝐞𝐫𝐧?💡**  
     
   The Circuit Breaker pattern is a software pattern that helps to 𝐩𝐫𝐞𝐯𝐞𝐧𝐭 𝐜𝐚𝐬𝐜𝐚𝐝𝐢𝐧𝐠 𝐟𝐚𝐢𝐥𝐮𝐫𝐞𝐬 𝐢𝐧 𝐚 𝐬𝐲𝐬𝐭𝐞𝐦 by introducing a layer of protection. This pattern is often used in distributed systems to provide a mechanism for failing fast, to prevent the system from becoming overwhelmed and unresponsive.  
     
   There 𝐚𝐫𝐞 𝐭𝐡𝐫𝐞𝐞 𝐬𝐭𝐚𝐭𝐞𝐬 that a circuit breaker can be in:  
     
   • 𝐂𝐥𝐨𝐬𝐞𝐝: In the closed state, the circuit breaker 𝐚𝐥𝐥𝐨𝐰𝐬 𝐫𝐞𝐪𝐮𝐞𝐬𝐭𝐬 𝐭𝐨 𝐩𝐚𝐬𝐬 through to the underlying system. That is the normal operating state for a circuit breaker.  
     
   • 𝐎𝐩𝐞𝐧: When the circuit breaker is in the open state, it 𝐬𝐭𝐨𝐩𝐬 𝐚𝐥𝐥 𝐫𝐞𝐪𝐮𝐞𝐬𝐭𝐬 from being sent to the underlying system. That is done to prevent further failures and to give the system a chance to recover.  
     
   • 𝐇𝐚𝐥𝐟-𝐎𝐩𝐞𝐧: In the half-open state, the circuit breaker 𝐚𝐥𝐥𝐨𝐰𝐬 𝐚 𝐥𝐢𝐦𝐢𝐭𝐞𝐝 𝐧𝐮𝐦𝐛𝐞𝐫 𝐨𝐟 𝐫𝐞𝐪𝐮𝐞𝐬𝐭𝐬 𝐭𝐨 𝐩𝐚𝐬𝐬 through to the underlying system. That is done to see if the system can handle requests again, without overwhelming it. If requests are successful, the circuit breaker will transition back to the closed state. If requests fail, the circuit breaker will return to the open state.  
     
   𝐑𝐞𝐚𝐥-𝐖𝐨𝐫𝐥𝐝 𝐄𝐱𝐚𝐦𝐩𝐥𝐞:  
     
   Let's say we have an e-commerce website that uses an external payment processing service.   
   The website sends requests to the payment processing service to charge customers' credit cards when they make a purchase.  
     
   If the payment processing service experiences a failure, such as a server outage, it may be unable to handle the requests from the e-commerce website. That could cause the website to become unresponsive or even crash.  
     
   We can prevent this by using a circuit breaker on an e-commerce website to protect itself from the payment processing service's failures.  
     
   The circuit breaker will monitor the payment processing service for a failure. If a failure is detected, it will trip the circuit breaker, transitioning it to the open state. That will stop the e-commerce website from sending requests to the payment processing service, preventing the website from becoming overwhelmed and unresponsive.  
     
   Once the payment processing service can recover, the circuit breaker can be reset to the half-open state. That will allow a limited number of requests to be sent to the payment processing service to see if it can handle them without failing again.  
     
   - If the requests are successful, the circuit breaker will transition back to the closed state.  
   - If the requests fail, the circuit breaker will return to the open state.  
     
   When we use a circuit breaker in this way, the e-commerce website can continue to operate, even in the face of failures in the payment processing service. That helps to make the website more resilient and fault-tolerant.
4. **Design Patterns 👨‍🏫  
     
   Adapter Design Pattern - Simplifed💡**  
     
   Definition:  
     
   The adapter pattern converts the interface of a class into another interface clients expect. The adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.  
     
   Real-Wolrd Analogy:  
     
   • Stefan from Serbia arrived in the UK.  
   • Stefan has a classic EU charger with a two-part outlet.  
   • The UK has a standard three-part plug.  
     
   How will Stefan charge his phone?  
     
   Answer: By using an adapter.  
     
   The adapter creates the possibility of connecting two different interfaces   
   (2 plugs).   
   It gets one type of input and produces a different type of output.  
     
   When to use:  
     
   • Ado .Net SqlAdapter, MySqlAdapter  
   • Allow a system to use classes of another system that is incompatible with it.
5. **What is CQRS? Simplified. 💡**Let's go to the kitchen, again. 😅  
     
   "𝐇𝐢, 𝐒𝐭𝐞𝐟𝐚𝐧. 𝐈 𝐤𝐞𝐞𝐩 𝐬𝐞𝐞𝐢𝐧𝐠 𝐩𝐨𝐬𝐭𝐬 𝐚𝐛𝐨𝐮𝐭 𝐂𝐐𝐑𝐒, 𝐛𝐮𝐭 𝐜𝐚𝐧 𝐲𝐨𝐮 𝐞𝐱𝐩𝐥𝐚𝐢𝐧 𝐢𝐧 𝐲𝐨𝐮𝐫 𝐰𝐚𝐲, 𝐢𝐧 𝐚 𝐬𝐢𝐦𝐩𝐥𝐢𝐟𝐢𝐞𝐝 𝐰𝐚𝐲, 𝐡𝐨𝐰 𝐢𝐭 𝐰𝐨𝐫𝐤𝐬?" - Sure.  
     
   So, CQRS stands for 𝐂𝐨𝐦𝐦𝐚𝐧𝐝 𝐐𝐮𝐞𝐫𝐲 𝐑𝐞𝐬𝐩𝐨𝐧𝐬𝐢𝐛𝐢𝐥𝐢𝐭𝐲 𝐒𝐞𝐠𝐫𝐞𝐠𝐚𝐭𝐢𝐨𝐧.  
   CQRS is an architectural pattern that separates the concerns of reading data from the concerns of updating data, by having different objects handle reading and updating data.  
     
   In .NET, we can implement CQRS using the 𝐌𝐞𝐝𝐢𝐚𝐭𝐑 𝐥𝐢𝐛𝐫𝐚𝐫𝐲, which provides a simple way to handle commands and queries in a decoupled way.  
     
   MediatR defines two main classes: 𝐈𝐑𝐞𝐪𝐮𝐞𝐬𝐭 and 𝐈𝐑𝐞𝐪𝐮𝐞𝐬𝐭𝐇𝐚𝐧𝐝𝐥𝐞𝐫.   
   • A class that implements IRequest represents the command.  
   • A class that implements IRequestHandler represents the query.  
     
   The MediatR library includes a mediator class that can be used to send commands and queries to the appropriate handlers.  
     
   Great. Okay, let's make it more clear.  
     
   𝐓𝐡𝐞 𝐚𝐧𝐚𝐥𝐨𝐠𝐲 𝐟𝐨𝐫 𝐂𝐐𝐑𝐒 𝐜𝐨𝐮𝐥𝐝 𝐛𝐞 𝐚 𝐜𝐡𝐞𝐟 𝐢𝐧 𝐚 𝐤𝐢𝐭𝐜𝐡𝐞𝐧.   
   The chef is responsible for executing commands (cooking orders) and querying the kitchen state (checking if ingredients are available).  
     
   The chef would have 𝐚 𝐬𝐞𝐩𝐚𝐫𝐚𝐭𝐞 𝐬𝐞𝐭 𝐨𝐟 𝐭𝐨𝐨𝐥𝐬 for:  
   • executing commands (a stove, oven)  
   • querying the kitchen state (a fridge, pantry).  
     
   The chef would have 𝐝𝐢𝐟𝐟𝐞𝐫𝐞𝐧𝐭 𝐬𝐤𝐢𝐥𝐥𝐬 𝐚𝐧𝐝 𝐤𝐧𝐨𝐰𝐥𝐞𝐝𝐠𝐞 for:  
   • executing commands (cooking techniques)  
   • querying the kitchen state (inventory management).  
     
   The chef would use 𝐝𝐢𝐟𝐟𝐞𝐫𝐞𝐧𝐭 𝐩𝐫𝐨𝐜𝐞𝐬𝐬𝐞𝐬 for:  
   • executing commands ( following a recipe)  
   • querying the state of the kitchen (checking inventory levels).  
     
   By separating the concerns of reading and updating data, the chef's work is more organized, manageable, and less prone to errors.  
   And the same applies to software systems that use CQRS, which makes the code more organized, and less prone to errors.
6. **𝐓𝐡𝐞 𝐑𝐞𝐩𝐨𝐬𝐢𝐭𝐨𝐫𝐲 𝐏𝐚𝐭𝐭𝐞𝐫𝐧 𝐰𝐢𝐭𝐡 𝐄𝐧𝐭𝐢𝐭𝐲𝐅𝐫𝐚𝐦𝐞𝐰𝐨𝐫𝐤 💡**  
     
     
   The repository pattern is a common way of organizing data access logic in an application. It provides a way to decouple the business logic of an application from the details of how data is stored and retrieved.  
     
   In the context of Entity Framework, the repository pattern can provide 𝐬𝐞𝐯𝐞𝐫𝐚𝐥 𝐛𝐞𝐧𝐞𝐟𝐢𝐭𝐬, including:  
     
   • A clean separation of concerns between the data access layer and the business logic layer of the application.  
   • Better testability, as repositories can be easily mocked or stubbed for testing purposes.  
   • Improved code readability and maintainability, as repositories can provide a more intuitive API for working with data.  
     
   To implement the repository pattern with Entity Framework, we would typically define an interface for the repository, which specifies the methods that should be available for working with data. This interface would then be implemented by a class that uses Entity Framework to perform the actual data access operations. Below is an example of what this might look like in C#.  
     
   𝐖𝐡𝐚𝐭 𝐚𝐛𝐨𝐮𝐭 𝐭𝐡𝐞 𝐜𝐨𝐧𝐬?  
     
   • One potential disadvantage of using the repository pattern with Entity Framework is that it 𝐜𝐚𝐧 𝐚𝐝𝐝 𝐚𝐧 𝐞𝐱𝐭𝐫𝐚 𝐥𝐚𝐲𝐞𝐫 𝐨𝐟 𝐜𝐨𝐦𝐩𝐥𝐞𝐱𝐢𝐭𝐲 to the application. This can make the code more difficult to understand and maintain, especially if the repositories don't provide a clear and intuitive API for working with data.  
     
   • The repository pattern can sometimes lead to inefficient queries being generated by Entity Framework. This can happen if the repository methods are not implemented most efficiently, or if the repository is used in a way that doesn't take advantage of the performance optimizations built into Entity Framework.  
     
   Some developers may argue that the repository pattern is unnecessary when using a modern ORM like Entity Framework, as it provides many of the benefits of the repository pattern out of the box. In these cases, using the repository pattern may be redundant or unnecessary.  
     
   Overall, whether or not the repository pattern is a good fit for our application will depend on the specific requirements and goals of the project. In some cases, the added complexity and potential for inefficiency may be outweighed by the benefits of a clean separation of concerns and improved testability. In other cases, it may be more appropriate to use Entity Framework directly without implementing the repository pattern.  
     
     
   Do you create repositories with EF? 📝👇  
   When?

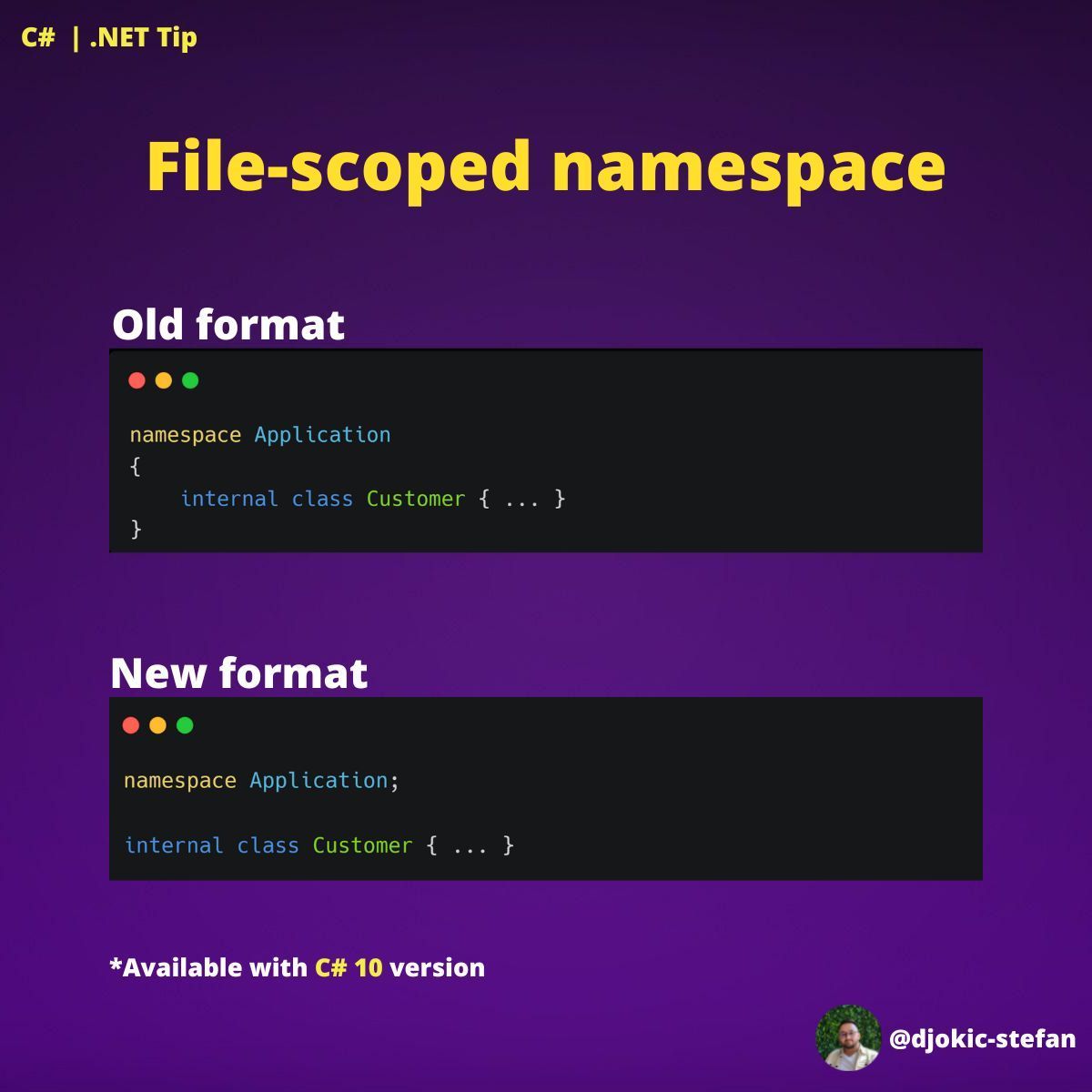


𝐐𝐮𝐢𝐜𝐤 𝐂# 𝐓𝐢𝐩𝐬:

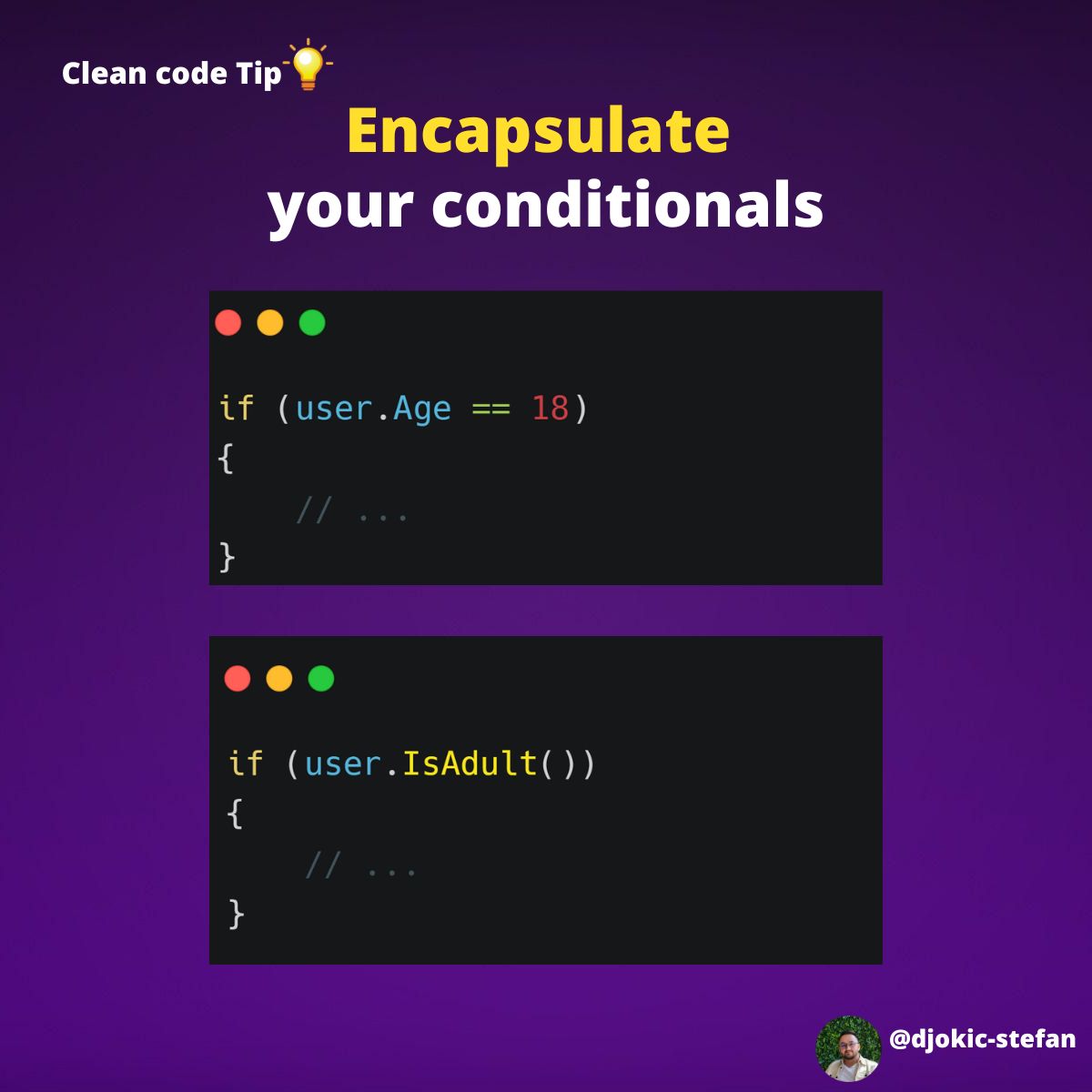
1. **Exception Handling Quick Tip 💡  
     
   Why you should not use 'throw *ex' in the Catch block?***  
     
   The answer is simple:  
     
   - throw; preserves the original stack trace of the exception, which is stored in the Exception.StackTrace property.  
   - throw ex; updates the StackTrace property of ex.  
     
   throw ex will reset the call stack in the exception to the point where this throw statement is losing the information about where the exception actually was created.



1. **I love this tiny feature! 😍  
     
   𝐅𝐢𝐥𝐞-𝐒𝐜𝐨𝐩𝐞𝐝 𝐧𝐚𝐦𝐞𝐬𝐩𝐚𝐜𝐞𝐬 🚀**  
     
   Starting with version C# 10, we can include a namespace as a statement, followed by a semi-colon, and without the curly brackets.  
     
   With the File-Scoped namespace, 𝐰𝐞 𝐫𝐞𝐦𝐨𝐯𝐞 𝐚 𝐥𝐞𝐯𝐞𝐥 𝐨𝐟 𝐧𝐞𝐬𝐭𝐢𝐧𝐠.   
   We can declare only one file-scoped namespace in the file.  
     
   We can 𝐚𝐮𝐭𝐨𝐦𝐚𝐭𝐞 the creation of file-scoped namespaces:  
     
   • Add 𝐍𝐞𝐰 𝐄𝐝𝐢𝐭𝐨𝐫𝐂𝐨𝐧𝐟𝐢𝐠 𝐟𝐢𝐥𝐞 (right-click on the project -> new)  
   • Set 𝐍𝐚𝐦𝐞𝐬𝐩𝐚𝐜𝐞 𝐝𝐞𝐜𝐥𝐚𝐫𝐚𝐭𝐢𝐨𝐧s to 𝐅𝐢𝐥𝐞-𝐒𝐜𝐨𝐩𝐞𝐝 from the 𝐂𝐨𝐝𝐞 𝐒𝐭𝐲𝐥𝐞 𝐭𝐚𝐛  
   • Create a new CS file, and you'll have a file-scoped namespace by default.  
     
   I started using file-scope namespaces early on.



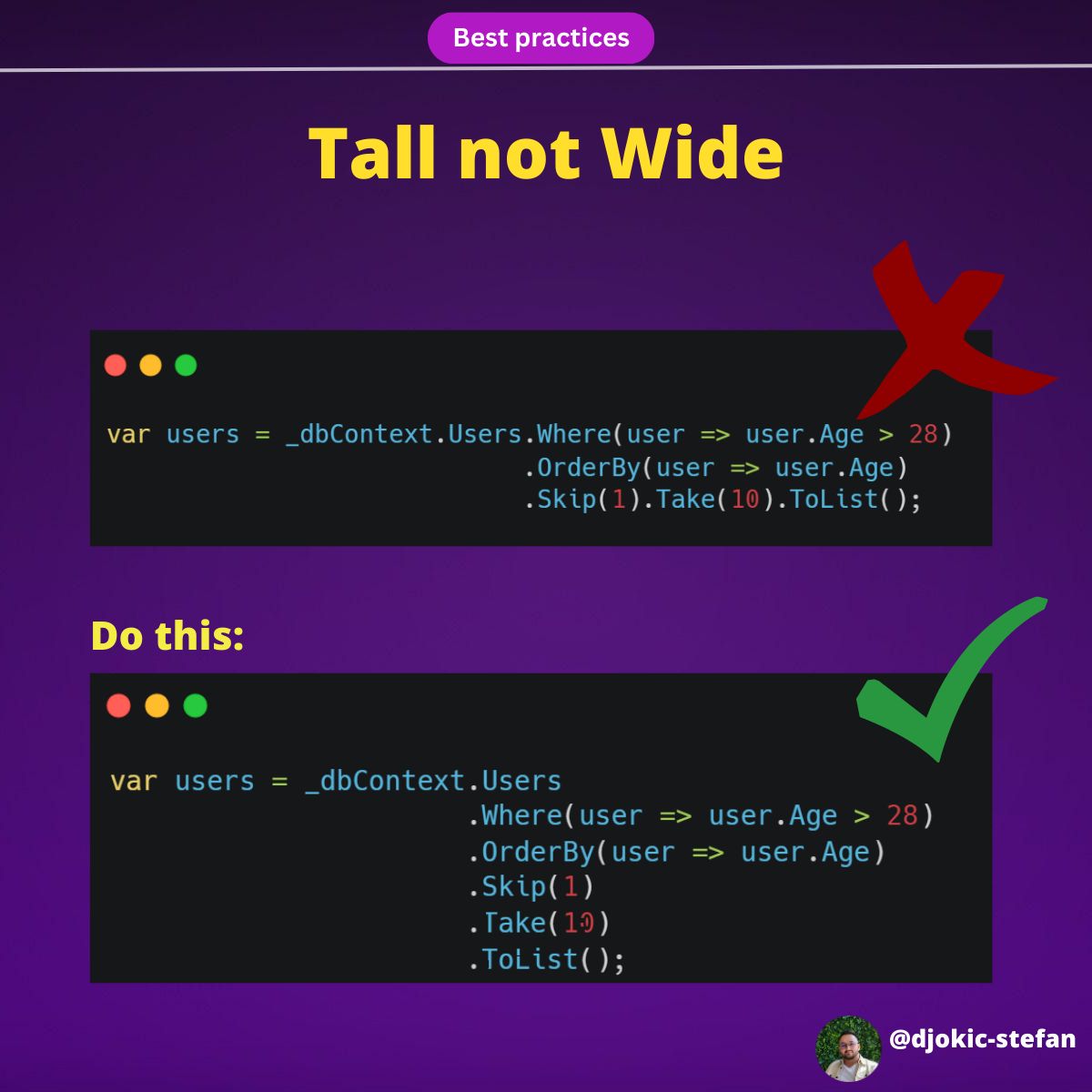
1. **Quick Clean Code Tip 💡**
2. **Encapsulate your conditionals ⚡**  
     
   Encapsulation is a way that helps isolate implementation details from the behavior exposed to clients of another class.  
     
   Also, encapsulation allows for greater control over the coupling of the code being written.  
     
   The real potential is seen in multiple conditionals.



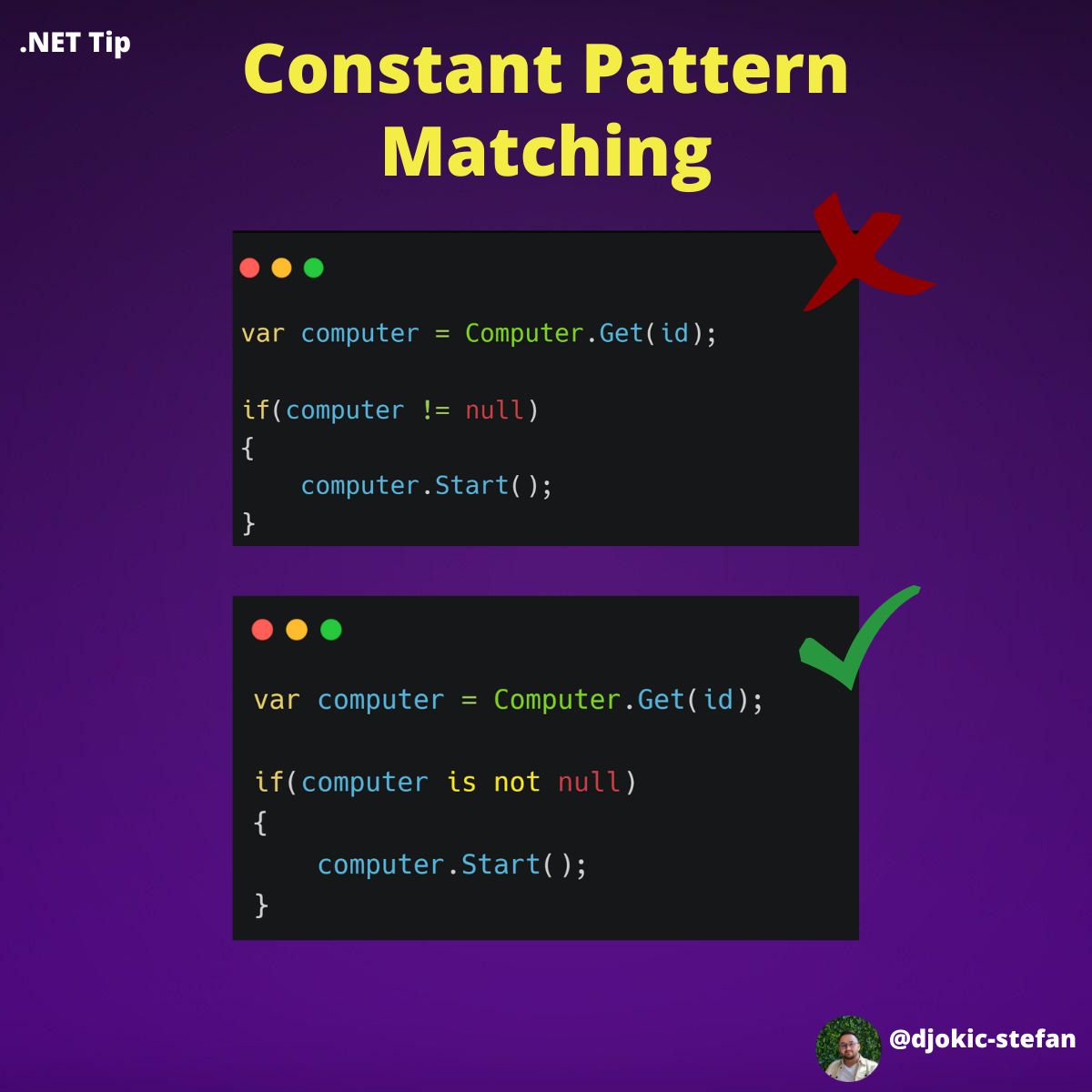
1. **Quick Clean Code Tip 💡  
     
   Don't repeat yourself.**  
     
   This error is common because we unconsciously repeat names or words.  
     
   This is bad practice in the long run.  
   It makes you have a very dirty code since there will be many things that will be repeated.  
     
   Fortunately, this is easy to fix.



1. **My favorite rule for writing queries.  
     
   "Write Tall not Wide" rule.**  
     
   This rule is simple but also very effective.  
     
   The "One dot per line" rule means: Use only one dot (.) in a single line of code to call the method.  
     
   Benefit: 𝐑𝐞𝐚𝐝𝐚𝐛𝐢𝐥𝐢𝐭𝐲.  
     
   It makes it easier to follow what are you doing.



1. **.NET Quick Tip 💡  
     
   How do you check if something is null?**  
     
   **Do you know about Constant Pattern Matching?**  
     
   We can use the constant pattern to test whether an expression is equal to a specified constant.  
     
   A popular use case for the constant pattern is null checking. This checks whether an object is null.  
     
   From C# 9 we can use the negation "is not" within the constant pattern.  
     
   Why I'm using this?  
     
   Both approaches will produce the same result, but the "is not null" syntax provides a more expressive and readable code.  
     
   It can be particularly helpful when dealing with complex expressions or when using pattern matching to check for specific types or conditions.  
     
   Also, the clear English language is explicit about what's going on.



1. **Quick C# | .NET Tip 💡  
     
   𝐃𝐨𝐧'𝐭 𝐝𝐨 𝐭𝐡𝐢𝐬!**  
     
   Null != Empty.  
     
   • Null - collection itself does not exist.  
   • Empty - collection exists with 0 elements  
     
   𝐖𝐞 𝐫𝐞𝐭𝐮𝐫𝐧 𝐍𝐮𝐥𝐥  
     
   • Possible NullReferenceException   
   • We always need to check for null  
   • Slow Performance (checking for null, throwing/catching an exception, etc.)  
     
   𝐖𝐞 𝐫𝐞𝐭𝐮𝐫𝐧 𝐚𝐧 𝐄𝐦𝐩𝐭𝐲 𝐜𝐨𝐥𝐥𝐞𝐜𝐭𝐢𝐨𝐧  
     
   • We don't need to check for null  
   • Good Performance (loop through the empty collection)  
     
   You should apply this to every collection type.

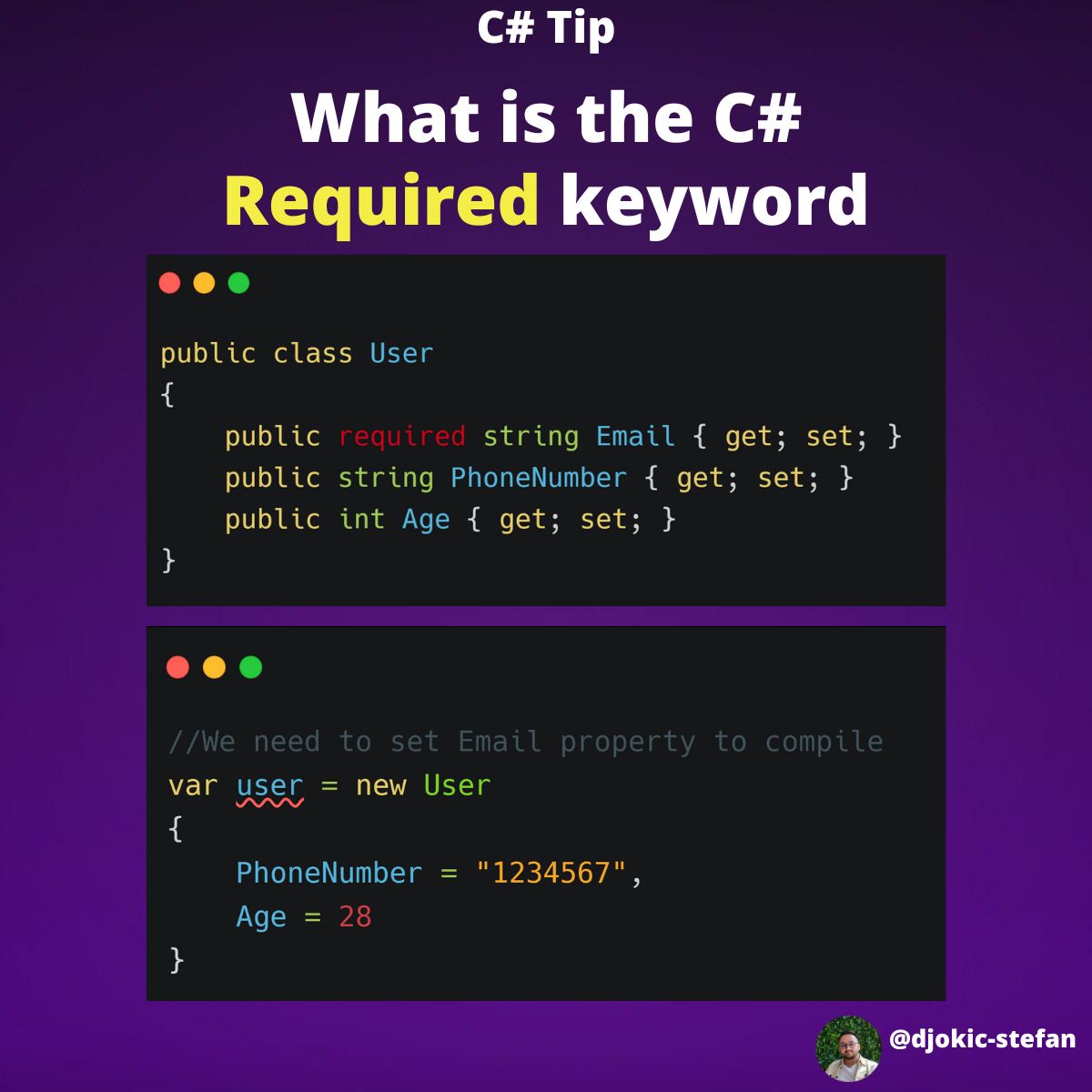


1. 𝐅𝐫𝐨𝐠 𝐰𝐢𝐭𝐡 𝐭𝐡𝐞 6 𝐥𝐞𝐠𝐬" problem 💡  
     
     
     
   𝐀: Why do you have a MyFrog class, when there is a standardized Frog class? Also, your frog has six legs?? 😮  
   𝐁: Haha, so you let me have it. 😅  
   𝐀: How to?  
   𝐁: You let me extend your standardized frog with extra legs. You allowed me to inherit your class.  
     
   **𝐀: I should have used the sealed keyword. 🤦‍♂️**  
     
   We are using Sealed classes to restrict the inheritance feature.   
   Once we define a class as a sealed class, we 𝐜𝐚𝐧𝐧𝐨𝐭 𝐢𝐧𝐡𝐞𝐫𝐢𝐭 𝐢𝐭 anymore.  
     
   sealed class SealedClass { }   
     
   We can also use the sealed keyword on a method or property that overrides a virtual method or property in a base class.    
   By doing this, we allow other classes to derive from the base class and 𝐩𝐫𝐞𝐯𝐞𝐧𝐭 𝐭𝐡𝐞𝐦 𝐟𝐫𝐨𝐦 𝐨𝐯𝐞𝐫𝐫𝐢𝐝𝐢𝐧𝐠 specific virtual methods or properties.  
     
   Why Sealed Classes?  
     
   • Take away the inheritance feature from the class users so they cannot derive a class from it.  
   • Best when we have a class with static members.   
   • Can lead to improved performances  
     
   Note:   
   The "Frog with the 6 legs" problem doesn't exist.   
   I came up with it while writing the post. 🤷‍♂️ 😂
2. **What are the 𝐝𝐢𝐟𝐟𝐞𝐫𝐞𝐧𝐜𝐞𝐬 𝐛𝐞𝐭𝐰𝐞𝐞𝐧 "𝐫𝐞𝐟" 𝐚𝐧𝐝 "𝐨𝐮𝐭" keywords?**  
     
     
   By default, the parameters are passed to a method 𝐛𝐲 𝐭𝐡𝐞 𝐯𝐚𝐥𝐮𝐞.  
   By using ref and out, we can pass a parameter 𝐛𝐲 𝐫𝐞𝐟𝐞𝐫𝐞𝐧𝐜𝐞.  
     
   At first glance, they seem very similar. And they did.   
   But there are differences.

𝐓𝐡𝐞 𝐫𝐞𝐟 𝐌𝐨𝐝𝐢𝐟𝐢𝐞𝐫:  
  
• Used to state that the parameter passed 𝒎𝒂𝒚 𝒃𝒆 modified by the method.  
• It is necessary the parameters should initialize before it passes to ref.  
• Useful when the called method also needs to change the value of a passed parameter.  
  
The out Modifier:  
  
• Used to state that the parameter passed 𝒎𝒖𝒔𝒕 𝒃𝒆 modified by the method.  
• It is not necessary to initialize parameters before they pass to an out.  
• Useful when a method returns multiple values.  
  
Note: Both ref and out parameters are treated the same at compile-time but different at run-time.  
  
These are just some of the differences.  
  
Do you know any more? 📝👇  
Are you using ref and out? 📝👇  
Where and when do you use them? 📝👇  
  
When do you use sealed?  📝👇



1. **What are the 𝐝𝐢𝐟𝐟𝐞𝐫𝐞𝐧𝐜𝐞𝐬 𝐛𝐞𝐭𝐰𝐞𝐞𝐧 "𝐫𝐞𝐟" 𝐚𝐧𝐝 "𝐨𝐮𝐭" keywords?**  
     
     
   By default, the parameters are passed to a method 𝐛𝐲 𝐭𝐡𝐞 𝐯𝐚𝐥𝐮𝐞.  
   By using ref and out, we can pass a parameter 𝐛𝐲 𝐫𝐞𝐟𝐞𝐫𝐞𝐧𝐜𝐞.  
     
   At first glance, they seem very similar. And they did.   
   But there are differences.  
     
   𝐓𝐡𝐞 𝐫𝐞𝐟 𝐌𝐨𝐝𝐢𝐟𝐢𝐞𝐫:  
     
   • Used to state that the parameter passed 𝒎𝒂𝒚 𝒃𝒆 modified by the method.  
   • It is necessary the parameters should initialize before it passes to ref.  
   • Useful when the called method also needs to change the value of a passed parameter.  
     
   The out Modifier:  
     
   • Used to state that the parameter passed 𝒎𝒖𝒔𝒕 𝒃𝒆 modified by the method.  
   • It is not necessary to initialize parameters before they pass to an out.  
   • Useful when a method returns multiple values.  
     
   Note: Both ref and out parameters are treated the same at compile-time but different at run-time.  
     
   These are just some of the differences.  
     
   Do you know any more? 📝👇  
   Are you using ref and out? 📝👇  
   Where and when do you use them? 📝👇
2. **What is the Required keyword in C#? 💡**  
     
   How could you use it?  
     
   It was introduced in C# 11.  
     
   When creating a new object instance, you can employ the "required" keyword to specify that a particular field or property must be initialized.  
     
   This keyword is especially useful when used alongside nullable reference types as it helps in creating a more robust design.  
     
   By utilizing the "required" keyword, if you attempt to create a new object instance but fail to assign a value to a required property, the compiler will generate an error during the compilation process.



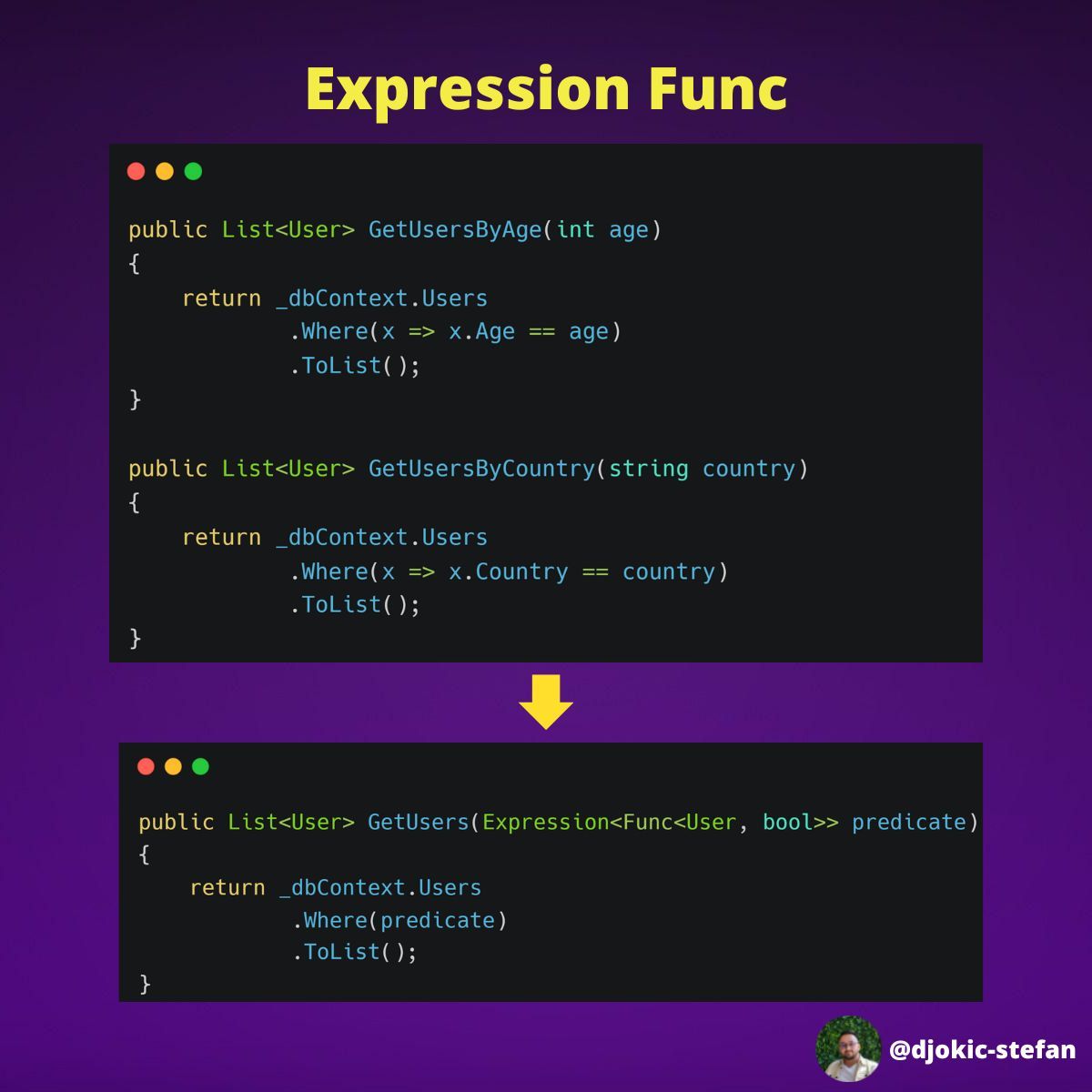
1. **What is API Versioning?  
     
   Why should you use it?**  
   API is software. Every piece of the software needs updates at some point.  
     
   When you are creating some updates on your API, you need to ensure that the changes don’t affect your API consumers (users). To ensure that, you need to introduce API versioning. It is the practice of managing changes to an API and ensuring that these changes will not disrupt users.  
     
   Why should you consider using it?  
     
   • Backward compatibility  
     
   It ensures that clients can continue using the older version while new clients can take advantage of the updated features.  
     
   • API evolution  
     
   As your API evolves over time, versioning allows you to introduce new features, deprecate outdated functionality, and make improvements without disrupting existing clients.  
     
   • Client flexibility  
     
   Different clients may have different requirements or may need specific features available in a particular version of the API.
2. **NET Tip 💡  
     
   How to deal with multiple HttpClient(s)?**  
   Your project requires interaction with multiple APIs.  
   Different Base addresses.  
   Different endpoints.  
     
   I'm using HttpClientFactory with TypedClient.  
     
   HttpClientFactory is designed to manage HttpClient instances, solving problems of socket exhaustion and providing better resource management.  
     
   Typed clients offer another level of abstraction. They encapsulate all interactions with the HttpClient within a class. This class is then injected wherever you need to perform HTTP requests, simplifying the code and making it easier to test.  
     
   How to add middleware to retrieve tokens for each API?  
   What if both APIs have the same Authorization?  
   How to add those things once?
3. **Quick C# Tip 💡  
     
   How do you typically define your DTOs?**  
     
   From C# 9 onwards, we have the option to utilize 𝗿𝗲𝗰𝗼𝗿𝗱𝘀 as a way to represent DTOs.  
     
   In the provided example, we use a primary constructor, enabling us to define the record in just one line of code.  
     
   Why?  
     
   Records are immutable by default, which means once a record object is created, it cannot be changed. This aligns well with the concept of DTOs, which are usually used to transfer states and should not be modified once created.



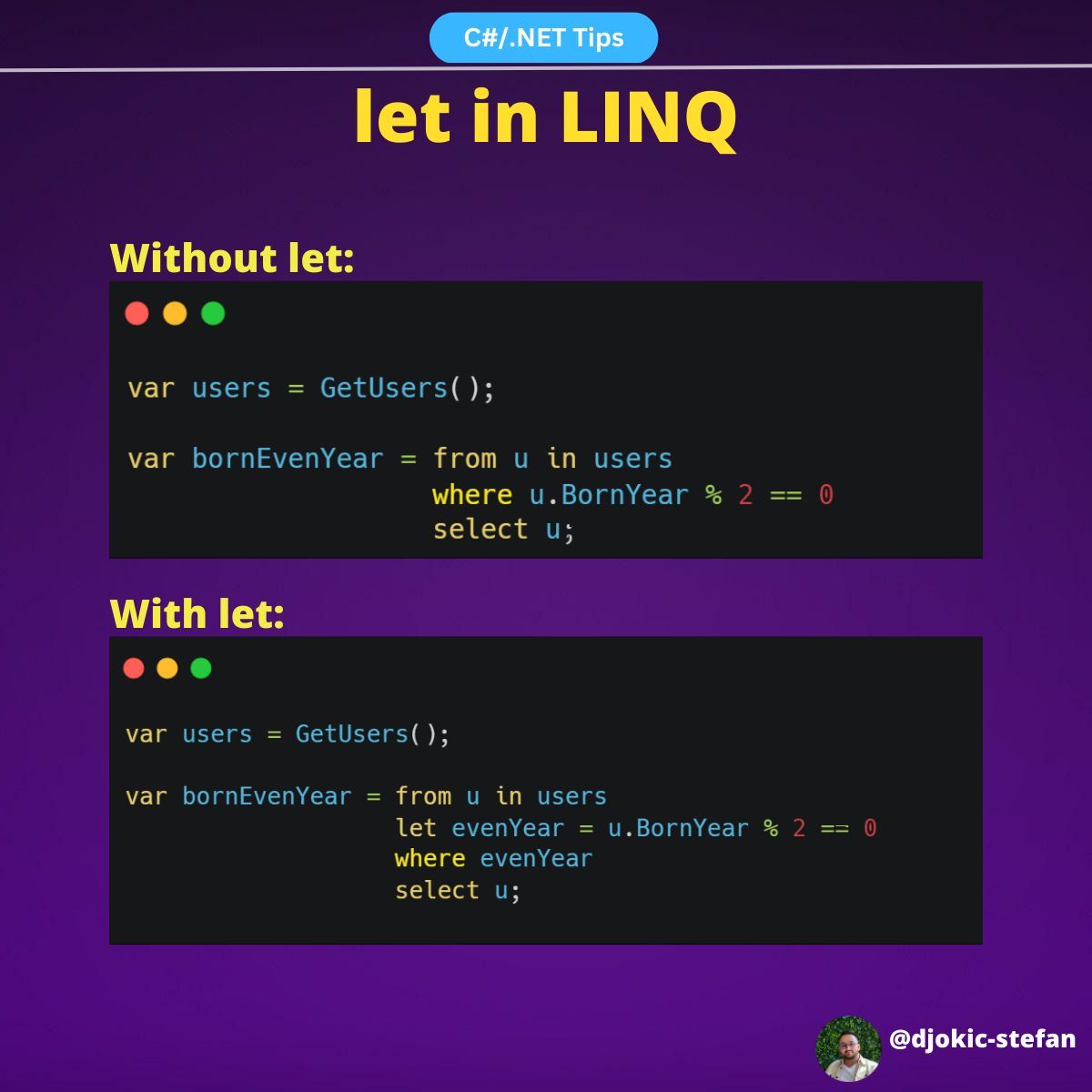
1. **What is a Webhook?  
     
   𝐒𝐢𝐦𝐩𝐥𝐢𝐟𝐢𝐞𝐝. 𝐄𝐱𝐚𝐦𝐩𝐥𝐞𝐬.**A webhook is a mechanism that allows a web application to send real-time notifications or data to another web application.  
     
   It is essentially a way for two different applications to communicate with each other in "real-time" rather than relying on periodic polling or manual data transfer.  
     
   Webhooks can be useful in a wide range of applications.  
     
   Some examples:  
     
   • Social media:   
   Receive notifications when new messages or comments are posted, or user profiles are updated.  
     
   • E-commerce:  
   Trigger actions like sending a confirmation email, updating inventory levels, or notifying shipping carriers when an order is placed.  
     
   • Collaboration tools:  
   Notify team members of new tasks, messages, or other updates in real-time  
     
   Have you had a chance to use webhooks somewhere?
2. **Juniors - pay attention! ❗❗❗  
     
   𝐃𝐞𝐩𝐞𝐧𝐝𝐞𝐧𝐜𝐲 𝐈𝐧𝐣𝐞𝐜𝐭𝐢𝐨𝐧 - 𝐒𝐢𝐦𝐩𝐥𝐢𝐟𝐢𝐞𝐝. 💡**  
     
   What is the definition of Dependency Injection?  
     
   A technique that allows us to remove hard-coded dependencies from our code. We can achieve this by 𝐢𝐧𝐣𝐞𝐜𝐭𝐢𝐧𝐠 𝐭𝐡𝐞 𝐝𝐞𝐩𝐞𝐧𝐝𝐞𝐧𝐜𝐢𝐞𝐬 𝐨𝐟 𝐚 𝐜𝐥𝐚𝐬𝐬 𝐭𝐡𝐫𝐨𝐮𝐠𝐡 𝐢𝐭𝐬 𝐜𝐨𝐧𝐬𝐭𝐫𝐮𝐜𝐭𝐨𝐫, 𝐫𝐚𝐭𝐡𝐞𝐫 𝐭𝐡𝐚𝐧 𝐡𝐚𝐫𝐝-𝐜𝐨𝐝𝐢𝐧𝐠 𝐭𝐡𝐞𝐦.  
     
   A real-world analogy for dependency injection would be a chef who prepares a meal.  
     
   • The chef has a recipe that specifies the ingredients needed to make the meal.   
   • Rather than the chef going to the grocery store to buy the ingredients themselves, the ingredients are provided to the chef by someone else.

• This allows the chef to focus on preparing the meal, rather than worrying about where to get the ingredients.  
  
In the same way:  
  
𝐃𝐞𝐩𝐞𝐧𝐝𝐞𝐧𝐜𝐲 𝐢𝐧𝐣𝐞𝐜𝐭𝐢𝐨𝐧 𝐚𝐥𝐥𝐨𝐰𝐬 𝐚 𝐜𝐥𝐚𝐬𝐬 𝐭𝐨 𝐟𝐨𝐜𝐮𝐬 𝐨𝐧 𝐢𝐭𝐬 𝐜𝐨𝐫𝐞 𝐟𝐮𝐧𝐜𝐭𝐢𝐨𝐧𝐚𝐥𝐢𝐭𝐲, 𝐫𝐚𝐭𝐡𝐞𝐫 𝐭𝐡𝐚𝐧 𝐰𝐨𝐫𝐫𝐲𝐢𝐧𝐠 𝐚𝐛𝐨𝐮𝐭 𝐰𝐡𝐞𝐫𝐞 𝐭𝐨 𝐠𝐞𝐭 𝐢𝐭𝐬 𝐝𝐞𝐩𝐞𝐧𝐝𝐞𝐧𝐜𝐢𝐞𝐬.  
  
In C# and .NET, we can use dependency injection by creating an interface that defines the contract for the dependent class and then implementing that interface in one or more concrete classes. Then, we can inject the interface into the class that depends on it, rather than hard-coding the concrete implementation.  
  
An example we can see on slides.  
  
In summary, dependency injection is a powerful technique that can help you write more flexible and maintainable code. Give it a try and see how it can benefit your projects!

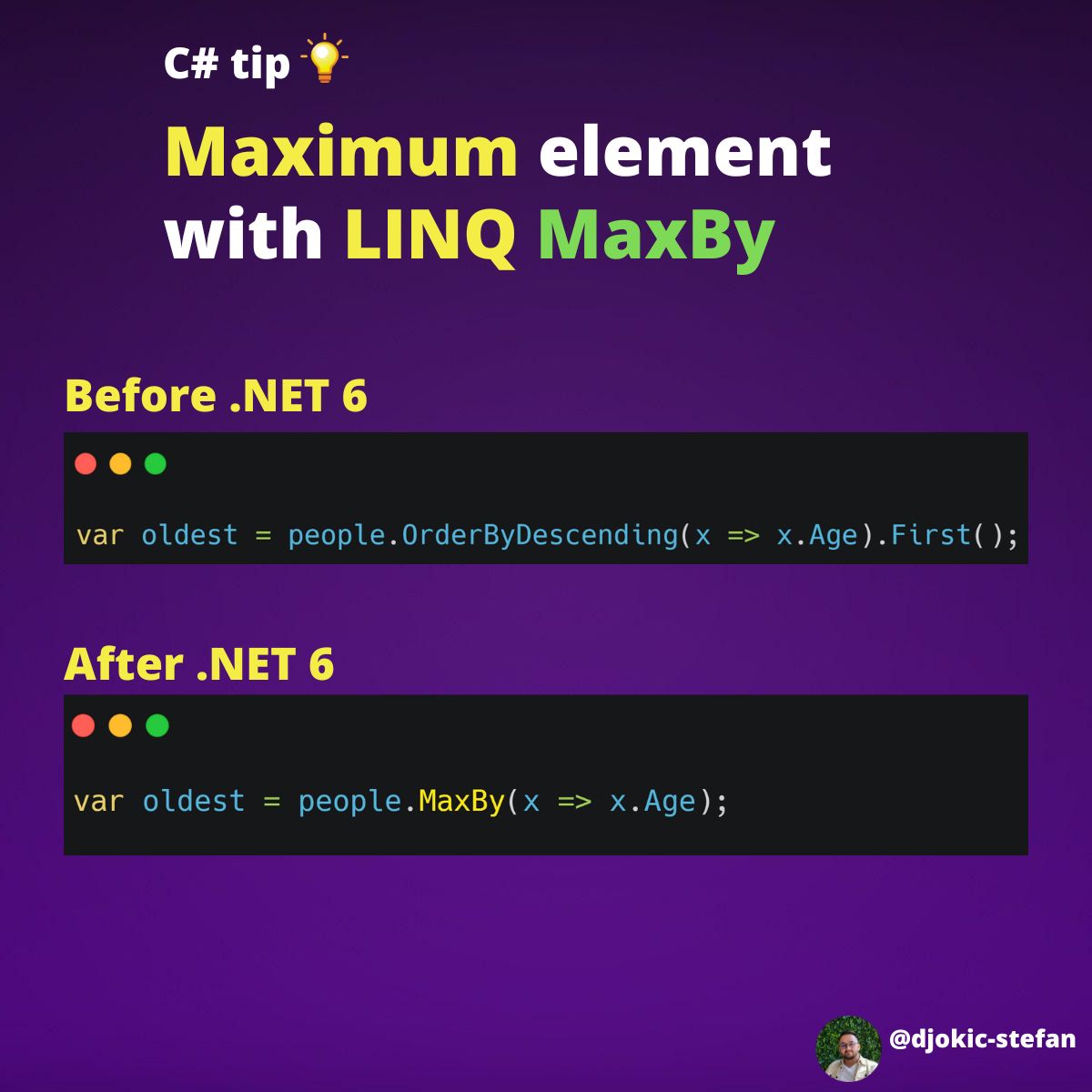
1. **What is 𝐄𝐱𝐩𝐫𝐞𝐬𝐬𝐢𝐨𝐧<𝐅𝐮𝐧𝐜<>>?**  
     
   Expression<Func<>> is an 𝐞𝐱𝐩𝐫𝐞𝐬𝐬𝐢𝐨𝐧 𝐭𝐫𝐞𝐞.  
     
   It’s a lambda expression in the form of a tree data structure.  
     
   It only holds information and composition of the contained lambda expression.  
     
   𝐄𝐱𝐚𝐦𝐩𝐥𝐞:  
     
   Expression<Func<int>> ten\_exp = () => 10;  
     
   The runtime engine can look “inside” Expression<Func<>> and translate it to another statement we need.
2. We often use Expression with EntityFramework.  
   𝐄𝐧𝐭𝐢𝐭𝐲𝐅𝐫𝐚𝐦𝐞𝐰𝐨𝐫𝐤 𝐰𝐢𝐥𝐥 𝐫𝐞𝐚𝐝 𝐭𝐡𝐞 𝐰𝐫𝐢𝐭𝐭𝐞𝐧 𝐄𝐱𝐩𝐫𝐞𝐬𝐬𝐢𝐨𝐧 𝐚𝐧𝐝 𝐠𝐞𝐧𝐞𝐫𝐚𝐭𝐞 𝐚𝐧 𝐒𝐐𝐋 𝐪𝐮𝐞𝐫𝐲 𝐛𝐚𝐬𝐞𝐝 𝐨𝐧 𝐢𝐭.  
     
   Instead of several different functions that work on the same Entity, we can define one with an Expression that we can use in several cases.  
     
   Do you use it?   
   Do you know the difference between Func<> and Expression<Funct<>>📝👇



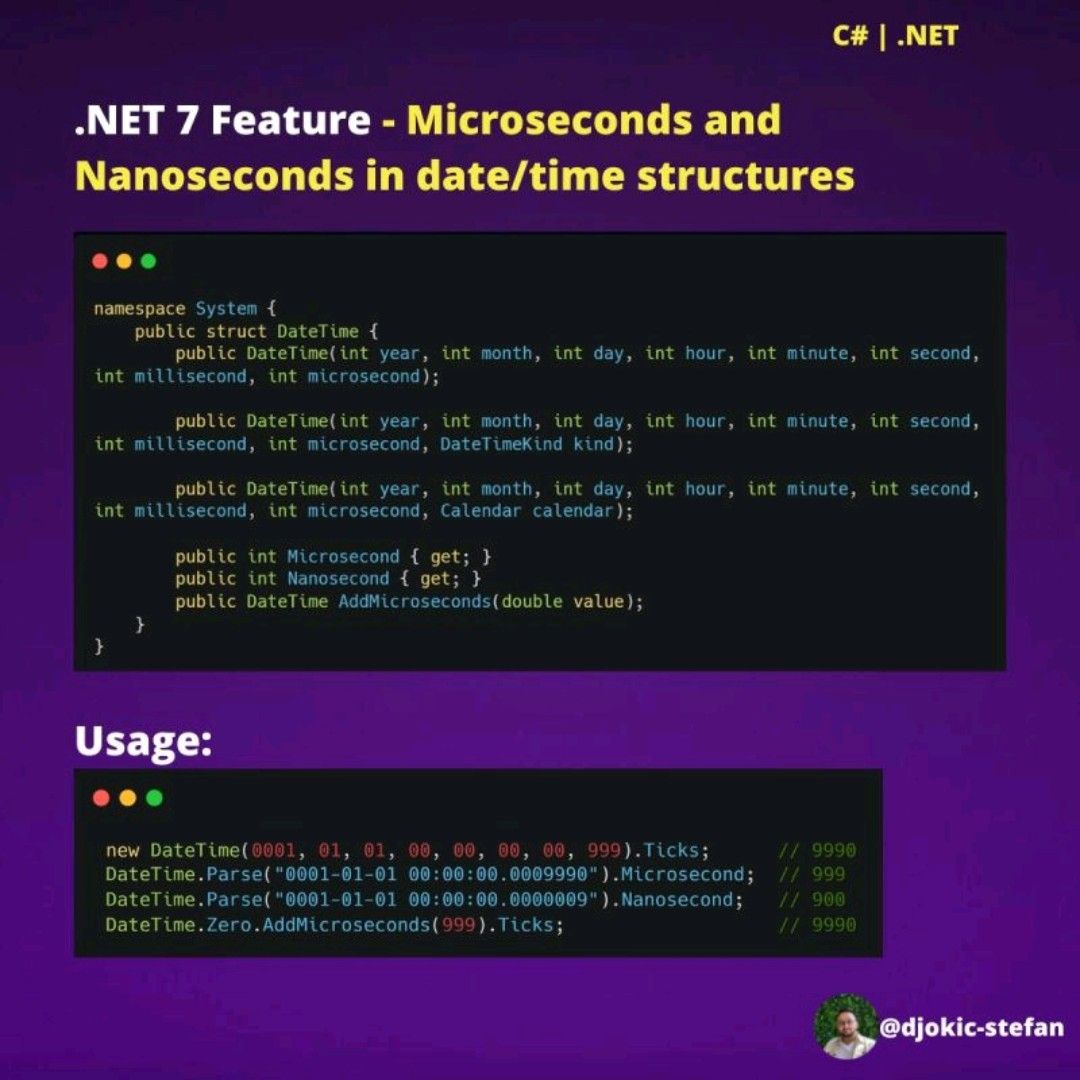
1. **Do you use local variables in LINQ?  
     
   Have you heard of the 𝐥𝐞𝐭 𝐤𝐞𝐲𝐰𝐨𝐫𝐝?**  
     
   Let keyword allows you to create a local variable at the LINQ query.  
     
   It is the same operation as building a local variable in a loop.  
     
   You can use let to create more complex expressions within a LINQ query. The query becomes more modular and easier to read, by breaking down a complex expression into smaller pieces and assigning them to temporary variables.  
     
   Using the "let" keyword can also help you optimize the performance of LINQ queries, as it can reduce the number of iterations required to process the data.  
     
   I'm giving you a simple example of using.  
   Did you have a chance to use it?  
   Did you face any benefits? 👇  
     
   On Monday I am sending my 23th issue of the newsletter with excellent content.



**How to find a maximum element with LINQ? 🤔**  
  
With .NET 6, they introduced the MaxBy method, specifically designed for this task.  
  
Before .NET 6 you couldn't do it like this.



1. **.NET 7 Feature 🔧  
     
   𝐌𝐧𝐝𝐬 𝐚𝐧𝐝 𝐍𝐚𝐧𝐨𝐬𝐞𝐜𝐨𝐧𝐝𝐬 𝐢𝐧 𝐝𝐚𝐭𝐞/𝐭𝐢𝐦𝐞 𝐬𝐭𝐫𝐮𝐜𝐭𝐮𝐫𝐞𝐬**  
     
   "The smallest time increment that could be used was the “tick” and its value is 100ns. The problem with this is that to determine a value in microseconds or nanoseconds you had to calculate everything based on the “tick” and this was not the most optimal thing in the world."  
     
   So they decided to add microseconds and nanoseconds to the different date and time structures that exist.  
     
   You can see the updated struct and usage examples below.

****

1. **Have you heard of The "𝐓𝐮𝐫𝐤𝐢𝐬𝐡 İ 𝐏𝐫𝐨𝐛𝐥𝐞𝐦"?  
     
   Don't use 𝐓𝐨𝐔𝐩𝐩()/𝐓𝐨𝐋𝐨𝐰𝐞𝐫() 𝐟𝐨𝐫 𝐬𝐭𝐫𝐢𝐧𝐠 𝐜𝐨𝐦𝐩𝐚𝐫𝐢𝐬𝐨𝐧 in all cultures. Why?**  
   Let me explain:  
     
   Let's imagine we have the word "interesting", and from the client, we got input ("INTERESTING").   
   We want to check if the input and our word are the same.  
     
   We use the 𝐓𝐨𝐔𝐩𝐩() method:  
     
   𝐄𝐧𝐠𝐥𝐢𝐬𝐡 𝐯𝐞𝐫𝐬𝐢𝐨𝐧: 𝐢.𝐓𝐨𝐔𝐩𝐩𝐞𝐫() => 𝐈  
     
   By comparing all letters we get the result: 𝐓𝐑𝐔𝐄.  
     
   𝐓𝐮𝐫𝐤𝐢𝐬𝐡 𝐯𝐞𝐫𝐬𝐢𝐨𝐧: 𝐢.𝐓𝐨𝐔𝐩𝐩() => İ  
     
   𝐈 is not equal to İ, so the result is 𝐅𝐀𝐋𝐒𝐄.  
     
   Conclusion: 𝐮𝐬𝐢 .𝐓𝐨𝐔𝐩𝐩𝐞𝐫() 𝐦𝐚𝐲 𝐫𝐞𝐭𝐮𝐫𝐧 𝐝𝐢𝐟𝐟𝐞𝐫𝐞𝐧𝐭 𝐫𝐞𝐬𝐮𝐥𝐭𝐬 𝐟𝐨𝐫 𝐝𝐢𝐟𝐟𝐞𝐫𝐞𝐧𝐭 𝐜𝐮𝐥𝐭𝐮𝐫𝐞𝐬.  
     
   Solution:  
     
   Use 𝐒𝐭𝐫𝐢𝐧𝐠𝐂𝐨𝐦𝐩𝐚𝐫𝐢𝐬𝐨𝐧.𝐎𝐫𝐢𝐠𝐢𝐧𝐚𝐥 or 𝐒𝐭𝐫𝐢𝐧𝐠𝐂𝐨𝐦𝐩𝐚𝐫𝐢𝐬𝐨𝐧.𝐎𝐫𝐢𝐠𝐢𝐧𝐚𝐥𝐈𝐠𝐧𝐨𝐫𝐞𝐂𝐚𝐬𝐞 𝐞𝐧𝐮𝐦𝐬 to compare by ordinal (binary) sort rules.  
     
   The question arises:  
     
   How many users from Turkey (or other countries with a similar problem) use the application? 🤔  
   Do I even need this?



internal modifier

The internal keyword is an access modifier for types and type members. We can declare a class as internal or its member as internal. Internal members are accessible only within files in the same assembly (.dll).

In other words, access is limited exclusively to classes defined within the current project assembly.

**Accessibility**

In same assembly (public)

* Can be accessed by objects of the class
* Can be accessed by derived classes

In other assembly (internal)

* Cannot be accessed by object
* Cannot be accessed by derived classes

## Indexes in SQL Server

SQL Indexes are used in relational databases to retrieve data quickly. They are similar to indexes at the end of the books whose purpose is quickly finding a topic. SQL provides Create Index, Alter Index, and Drop Index commands used to create a new index, update an existing one, and delete an index in [SQL Server.](https://www.c-sharpcorner.com/topics/sql-server)

* Data is internally stored in a SQL Server database in "pages" where the size of each page is 8KB.
* A continuous eight pages are called an "Ex. "nt."
* When we create the table, one extent will be allocated for two tables; when it is computed, it is filled with the data. Then another time will be given, and this extent may or may not be continuous to the first extent.

### **Type of Indexes**

SQL Server supports two types of indexes:

1. Clustered Index
2. Non-Clusterd Index.

### **Clustered Index in SQL Server**

A B-Tree (computed) clustered index is the Index that will arrange the rows physically in the memory in sorted order.

An advantage of a clustered index is that searching for a range of values will be fast. A clustered index is internally maintained using a B-Tree data structure leaf node of the btree of the clustered Index will contain the table data; you can create only one clustered Index for a table.

### **Nonclustered Index in SQL Server**

* A nonclustered index is an index that will not arrange the rows physically in the memory in sorted order.
* An advantage of a nonclustered index is that searching for the values in a range will be fast.
* You can create a maximum of 999 nonclustered indexes on a table, 254 up to SQL Server 2005.
* A nonclustered index is also maintained in a B-Tree data structure. Still, leaf nodes of a B-Tree of the nonclustered Index contain the pointers to the pages that contain the table data and not the table data directly.

### **Abstract Class**

We can not create an object of an abstract class and can call the method of abstract class with the help of class name only.

Take a look at an Abstract class example,

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace ConsoleApplication4

{

abstract class M1

{

public int add(int a, int b)

{

return (a + b);

}

}

class M2 :M1

{

public int mul(int a, int b)

{

return a \* b;

}

}

class test

{

static void Main(string[] args)

{

M2 ob = new M2();

int result = ob.add(10, 20);

Console.WriteLine("the result is {0}", result);

Console.ReadLine();

}

}

}

### **An Interface**

The syntax of an Interface looks like this,

interface{     //Interface member}

**Note**

1. An Interface member cannot contain code bodies.
2. Type definition members are forbidden.
3. Properties are defined in an interface with the help of an access block get and set, which are permitted for the property.

interface myInterface

{

int myint

{

get;

set;

}

}

C#

Copy

Take a look in an interface example,

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace ConsoleApplication3

{

interface MyInterface

{

void myMethod();

}

class MyClass : MyInterface

{

public static void Main()

{

MyClass cls = new MyClass();

cls.myMethod();

}

public void myMethod()

{

Console.WriteLine("welcome to MCN IT SOLUTION");

Console.ReadLine();

}

}

}

Polymorphism means “Many Forms”. In polymorphism, poly means “Many,” and morph means “Forms.” polymorphism is one of the main pillars in Object Oriented Programming. It allows you to create multiple methods with the same name but different signatures in the same class. The same name methods can also be in derived classes.

There are two types of polymorphism,

1. Method Overloading
2. Method Overriding

In this article, I will explain the method overloading and method overriding concept in C#. Furthermore, I will try to demonstrate step-by-step differences between these.

Method Overloading

Method Overloading is a type of polymorphism. It has several names like “Compile Time Polymorphism” or “Static Polymorphism,” and sometimes it is called “Early Binding”.

Method Overloading means creating multiple methods in a class with the same names but different signatures (Parameters). It permits a class, struct, or interface to declare multiple methods with the same name with unique signatures.

The compiler automatically calls the required method to check the number of parameters and their type passed into that method.

using System;

namespace DemoCsharp

{

class Program

{

public int Add(int num1, int num2)

{

return (num1 + num2);

}

public int Add(int num1, int num2, int num3)

{

return (num1 + num2 + num3);

}

public float Add(float num1, float num2)

{

return (num1 + num2);

}

public string Add(string value1, string value2)

{

return (value1 + " " + value2);

}

static void Main(string[] args)

{

Program objProgram = new Program();

Console.WriteLine("Add with two int parameter :" + objProgram.Add(3, 2));

Console.WriteLine("Add with three int parameter :" + objProgram.Add(3, 2, 8));

Console.WriteLine("Add with two float parameter :" + objProgram.Add(3 f, 22 f));

Console.WriteLine("Add with two string parameter :" + objProgram.Add("hello", "world"));

Console.ReadLine();

}

}

}

C#

Copy

In the above example, you can see four methods with the same name, but the type of parameters or several parameters is different. So when you call Add(4,5), the compiler automatically calls the method, which has two integer parameters. On the other hand, when you call Add(“hello”,”world”), the compiler calls the method, which has two string parameters. So basically, in method overloading compiler checks, which method should be called at the compilation time.

**Note:**Changing the method's return type does not overload the method. You cannot create a method overloaded to vary only by return type.

Method Overriding

Method Overriding is a type of polymorphism. It has several names like “Run Time Polymorphism” or “Dynamic Polymorphism,” and sometimes it is called “Late Binding”.

Method Overriding means having two methods with the same name and same signatures [parameters]; one should be in the base class, and another method should be in a derived class [child class]. You can override the functionality of a base class method to create the same name method with the same signature in a derived class. You can achieve method overriding using inheritance. Virtual and Override keywords are used to achieve method overriding.

using System;

namespace DemoCsharp

{

class BaseClass

{

public virtual int Add(int num1, int num2)

{

return (num1 + num2);

}

}

class ChildClass: BaseClass

{

public override int Add(int num1, int num2)

{

if (num1 <= 0 || num2 <= 0)

{

Console.WriteLine("Values could not be less than zero or equals to zero");

Console.WriteLine("Enter First value : ");

num1 = Convert.ToInt32(Console.ReadLine());

Console.WriteLine("Enter First value : ");

num2 = Convert.ToInt32(Console.ReadLine());

}

return (num1 + num2);

}

}

class Program

{

static void Main(string[] args)

{

BaseClass baseClassObj;

baseClassObj = new BaseClass();

Console.WriteLine("Base class method Add :" + baseClassObj.Add(-3, 8));

baseClassObj = new ChildClass();

Console.WriteLine("Child class method Add :" + baseClassObj.Add(-2, 2));

Console.ReadLine();

}

}

}

C#

Copy

In the above example, I have created two same-name methods in the BaseClass and the ChildClass. When you call the BaseClass Add method with less than zero value as parameters, then it adds successfully. But when you call the ChildClass Add method with less than zero value, it checks for a negative value. If the passing values are negative, it asks for a new value.

So, here it is clear that we can modify the base class methods in derived classes.

Points to be remembered,

1. The method cannot be private.
2. Only abstract or virtual methods can be overridden.
3. Which method should be called is decided at run time.

The two most common types of authentication are **session cookies and tokens.**

Session Cookies

* Session Cookies are small data stored both on the server and the client.
* The server often keeps track of sessions in a database or memory.
* A browser controls cookies on the client side. They're included with every request.
* Session cookies are stateful.
* Each session has a unique ID that the server uses to identify the current user and all the information related to that user.
* Cookies allow a domain and its subdomains to exchange information.
* Sharing cookie information with another domain is not possible.
* You can use the "**HttpOnly**" setting to prevent JavaScript tampering on client sites.
* Remember that only HTTPS connections are secure for cookies. The "**Secure**" flag can be used for this reason. It ensures that cookies will be sent only if the connection type is HTTPS.

Tokens

* In essence, tokens are a collection of letters and numbers.
* Tokens have no state. It implies that no information about the token needs to be stored on the server.
* The tokens stand alone. This signifies that the token has all the data needed for server-side verification.
* Since no database searches are necessary, they are appropriate for API authentication.
* Tokens are incredibly adaptable and work with multiple platforms.
* Additionally, there is nothing like domain restriction. Tokens can be transferred between various domains. Because they are self-contained, they are larger in size than cookies.

Session is a server side object, which transfer or access data between page call.  
Cookies is a object which is client side/client machine which store some text information of browser and server.

The data that a user enters into multiple pages of a website can be saved using a combination of [**cookies**](https://www.tutorialspoint.com/internet_security/internet_security_cookies.htm) and sessions. Cookies and sessions are both vitally important since they record the data that the user has provided for a variety of purposes.

Cookies and Sessions are used to store information. Cookies are only stored on the client-side machine, while sessions get stored on the client as well as the server.

Read through this article to find out more about cookies and sessions and how they are different from each other.

## What is a Cookie?

Cookies are little text-based files that are kept on the user's computer and are accessible only by that user's browser. It is possible for a cookie's size to reach a maximum of 4 KB. Cookies are also referred to as [**HTTP**](https://www.tutorialspoint.com/http/http_overview.htm) cookies, online cookies, and internet cookies, amongst other names.

When a person signs into a website or application for the very first time, that website transmits a large amount of information to the user's computer in the form of cookies.

When a user opens a site, these cookies maintain track of all of the activities and surfing they do on the site. The information is saved using the string type. The information that is gathered by cookies is not considered to be secure because the information is kept in texts that are simple to read and understand. The users have the ability to control whether the cookies are enabled or disabled at any time they see fit. Only the user who initially created a cookie has access to the cookie's information; no other users can see it.

Cookies are created with the help of an [**HTTP header**](https://www.tutorialspoint.com/http/http_header_fields.htm) and then transferred between the browser and the server. Cookies are saved in the Temporal Internet File Folder by Internet Explorer, but the location where they are saved is determined by the browser being used. Cookies read the data that is generated when we search for and play certain songs on YouTube. Then, the next time that we open YouTube, the same songs or recommendations that are similar to them are displayed.

### **Why Use Cookies?**

Since HTTP is a stateless protocol, it does not save any information about users on its servers. Cookies are a useful tool for accomplishing this goal. It makes it possible for us to save the information on the user's computer and monitor the state of any apps that are being used.

## What is a Session?

Sessions are used to temporarily store data on the server so that it can be utilized on a variety of different websites. To put it more simply, a session is the amount of time that a user spends engaged in a particular activity. This period of time begins when the user accesses any website or program and continues until the user exits the application or shuts down the computer.

Because the HTTP protocol does not keep track of the user's status, the web server does not recognize the user as they are navigating the apps available on the internet. The information that the user enters into the program on the home page or any other page will not be moved to any other pages in the application. Sessions are what are used to remove this limitation from the game. Session variables, which can be of any sort, are where the user's data can be saved if they are chosen to do so. The data that is stored by session variables is either encrypted or converted to a binary form on the server, which protects the data from being accessed by a third party.

When the user of the application logs out of that application or closes down their computer, the session value is automatically detached. It is necessary for the session values to be saved in the database if they are to be retained for a longer period of time. Every single session is unique to the individual user, and there is no limit to the number of sessions that can be used within a single application because there are no constraints placed on it. A user can be recognized with the assistance of a session, which is a one-of-a-kind number that is kept on the server in the form of a cookie, a form field, or a URL.

The Session ID, which is a one-of-a-kind number that is temporarily stored on the server, is what is used to identify the user. It is either a cookie, a form field, or a URL that is saved.

### **Why Use Sessions?**

Sessions are used to store information such as User ID over the server more securely, where it cannot be altered. This prevents the information from being tampered with.

In addition to this, sessions can transfer the information from one web page to another in the form of value.

Sessions can be used as a substitute for cookies in web browsers that do not support cookies, allowing for the storage of variables in a manner that is more impenetrable.

## Comparison between Cookie and Session

The following table highlights the major differences between a cookie and a session −

| **Basis of Comparison** | **Cookie** | **Session** |
| --- | --- | --- |
| Definition | Cookies are client-side files that are stored on a local computer and contain user information. | Sessions are server-side files that store user information. |
| Expiry | Cookies expire after the user specified lifetime. | The session ends when the user closes the browser or logs out of the program. |
| Data storage | It can only store a limited amount of data. | It is able to store an unlimited amount of information. |
| Capacity | Cookies can only store up to a maximum of 4 KB of data in a browser. | There is a maximum memory restriction of 128 megabytes that a script may consume at one time. However, we are free to maintain as much data as we like within a session. |
| Function | It is not necessary for us to execute a function in order to get cookies going because they are stored on the local computer. | Utilizing the session start()method is required before we can begin the session. |
| Data Format | Cookies are used to store information in a text file. | The data is saved in an encrypted format during sessions. |
| Storage | Cookies are stored on a limited amount of data. | A session can store an unlimited amount of data. |