Emner til eksamen d. 7. januar 2021

Systemudvikling:

# Modellering (Objekt- og domænemodel, use cases, SD, SSD, OC, DCD, Wireframe)

## SCRUM:

# Kvalitet:

# Sporbarhed:

# Brugergrænseflade og brugertest:

# Forretningsmodellering (BPMN):

# Sikkerhed (GDPR):

Persondataloven

# Forandringsledelse:

# Modellering (OM/DM/UC):

Programming:

# Principles and patterns:

## Computational Thinking:

(På dansk Nedbrydning, mønstergenkendelse, abstraktion, algoritmer)

* Decomposition - breaking down a complex problem or system into smaller, more manageable parts
* Pattern recognition – looking for similarities among and within problems
* Abstraction – focusing on the important information only, ignoring irrelevant detail
* Algorithms - developing a step-by-step solution to the problem, or the rules to follow to solve the problem

Computational thinking involves taking that complex problem and breaking it down into a series of small, more manageable problems (decomposition). Each of these smaller problems can then be looked at individually, considering how similar problems have been solved previously (pattern recognition) and focusing only on the important details, while ignoring irrelevant information (abstraction). Next, simple steps or rules to solve each of the smaller problems can be designed (algorithms).

Finally, these simple steps or rules are used to program a computer to help solve the complex problem in the best way.

## Encapsulation (OOP):

In an Object-Oriented programming language, encapsulation is one of the key language features. Encapsulation is the procedure of encapsulating data and functions into a single unit (called class).

The need of encapsulation is to protect or prevent the code (data) from accidental corruption due to the silly little errors that we are all prone to make. In Object oriented programming data is treated as a critical element in the program development and data is packed closely to the functions that operate on it and protects it from accidental modification from outside functions.

Rather than defining the data in the form of public, we can declare those fields as private. The Private data are manipulated indirectly by two ways:

* Encapsulation via accessors and mutators, by using Get and Set methods for a private field.
* Encapsulation via properties, which is a cleaner version of using Get/Set methods.

You can even use Auto-Properties which creates its own private field automatically, so you do not have to.

## Polymorphism (OOP):

### Compile Time Polymorphism:

In c#, Compile Time Polymorphism means defining multiple methods with the same name but with different parameters. By using compile-time polymorphism, we can perform different tasks with the same method name by passing different parameters.

In c#, the compile-time polymorphism can be achieved by using method overloading and it is also called early binding or static binding.

### Run Time Polymorphism:

In c#, Run Time Polymorphism means overriding a base class method in the derived class by creating a similar function and this can be achieved by using override & virtual keywords along with inheritance principle.

By using run-time polymorphism, we can override a base class method in the derived class by creating a method with the same name and parameters to perform a different task.

In c#, the run time polymorphism can be achieved by using method overriding and it is also called late binding or dynamic binding.

## Inheritance (OOP):

Inheritance, together with encapsulation and polymorphism, is one of the three primary characteristics of object-oriented programming. Inheritance enables you to create new classes that reuse, extend, and modify the behavior defined in other classes. The class whose members are inherited is called the base class, and the class that inherits those members is called the derived class. A derived class can have only one direct base class. However, inheritance is transitive. If ClassC is derived from ClassB, and ClassB is derived from ClassA, ClassC inherits the members declared in ClassB and ClassA.

Conceptually, a derived class is a specialization of the base class. For example, if you have a base class Animal, you might have one derived class that is named Mammal and another derived class that is named Reptile. A Mammal is an Animal, and a Reptile is an Animal, but each derived class represents different specializations of the base class.

When you define a class to derive from another class, the derived class implicitly gains all the members of the base class, except for its constructors and finalizers. The derived class reuses the code in the base class without having to reimplement it. You can add more members in the derived class. The derived class extends the functionality of the base class.

Both interfaces and classes can be inherited / implemented. A class can only inherit from one other class, but it can implement any number of interfaces.

## Abstraction (OOP):

## SOLID:

There are five SOLID principles:

* Single Responsibility Principle (SRP)
* Open Closed Principle (OCP)
* Liskov Substitution Principle (LSP)
* Interface Segregation Principle (ISP)
* Dependency Inversion Principle (DIP)

SRP:

A class should not be loaded with multiple responsibilities and a single responsibility should not be spread across multiple classes or mixed with other responsibilities. The reason is that more changes requested in the future, the more changes the class need to apply.

OCP:

This principle suggests that the class should be easily extended but there is no need to change its core implementations.

The application or software should be flexible to change. How change management is implemented in a system has a significant impact on the success of that application/ software. The OCP states that the behaviors of the system can be extended without having to modify its existing implementation.

i.e., New features should be implemented using the new code, but not by changing existing code. The main benefit of adhering to OCP is that it potentially streamlines code maintenance and reduces the risk of breaking the existing implementation.

LSP:

LSP states that the child class should be perfectly substitutable for their parent class. If class C is derived from P then C should be substitutable for P.

We can check using LSP that inheritance is applied correctly or not in our code.

LSP is a fundamental principle of SOLID Principles and states that if program or module is using base class then derived class should be able to extend their base class without changing their original implementation.

ISP:

Interface segregation principle is required to solve the design problem of the application. When all the tasks are done by a single class or in other words, one class is used in almost all the application classes then it has become a fat class with overburden. Inheriting such class will results in having sharing methods which are not relevant to derived classes but its there in the base class so that will inherit in the derived class.

Using ISP, we can create separate interfaces for each operation or requirement rather than having a single class to do the same work.

DIP:

The principle says that high-level modules should depend on abstraction, not on the details, of low-level modules. In simple words, the principle says that there should not be a tight coupling among components of software and to avoid that, the components should depend on abstraction.

The terms Dependency Injection (DI) and Inversion of Control (IoC) are generally used as interchangeably to express the same design pattern. The pattern was initially called IoC, but Martin Fowler (known for designing the enterprise software) anticipated the name as DI because all frameworks or runtime invert the control in some way and he wanted to know which aspect of control was being inverted.

Inversion of Control (IoC) is a technique to implement the Dependency Inversion Principle in C#. Inversion of control can be implemented using either an abstract class or interface. The rule is that the lower level entities should join the contract to a single interface and the higher-level entities will use only entities that are implementing the interface. This technique removes the dependency between the entities.

## Layered Architecture:

## Grasp (low coupling, high cohesion, controller) (OOP):

**General Responsibility Assignment Software Patterns.**

## MVVM:

## Observer:

## Repository:

## Singleton:

A singleton design pattern ensures a class has only one instance in the program and provides a global point of access to it. A singleton is a class that only allows a single instance of itself to be created and usually gives simple access to that instance. Most commonly, singletons do not allow any parameters to be specified when creating the instance since the second request of an instance with a different parameter could be problematic. If the same instance should be accessed for all requests with the same parameter, then the factory pattern is more appropriate.

There are various ways to implement a singleton pattern in C#. The following are the common characteristics of a singleton pattern:

* A single constructor, that is private and parameterless.
* The class is sealed.
* A static variable that holds a reference to the single created instance, if any.
* A public static means of getting the reference to the single created instance, creating one if necessary.

# Programming:

## Program:

A program consists of a set of instructions that a computer can interpret and execute. A program must have a main method.

## Class:

A type that is defined as a class is a reference type. At run time, when you declare a variable of a reference type, the variable contains the value null until you explicitly create an instance of the class by using the new operator or assign it an object of a compatible type that may have been created elsewhere.

Classes are declared by using the class keyword followed by a unique identifier. The remainder of the definition is the class body, where the behavior and data are defined. Fields, properties, methods, and events on a class are collectively referred to as class members and are private by default.

In general, class declarations can include these components, in order:

* Modifiers: A class can be public or internal etc. By default, modifier of class is internal.
* Keyword class: A class keyword is used to declare the type class.
* Class Identifier: The variable of type class is provided. The identifier (or name of class) should begin with a initial letter which should be capitalized by convention.
* Base class or Super class: The name of the class’s parent (superclass), if any, preceded by the: (colon). This is optional.
* Interfaces: A comma-separated list of interfaces implemented by the class, if any, preceded by the: (colon). A class can implement more than one interface. This is optional.
* Body: The class body is surrounded by {} (curly braces).

Constructors in class are used for initializing new objects. Fields are variables that provide the state of the class and its objects, and methods are used to implement the behavior of the class and its objects.

## Object:

Although they are sometimes used interchangeably, a class and an object are different things. A class defines a type of object, but it is not an object itself. An object is a concrete entity based on a class and is sometimes referred to as an instance of a class.

Objects can be created by using the new keyword followed by the name of the class that the object will be based on, like this:

Customer object1 = new Customer();

## Field:

A field is a variable of any type that is declared directly in a class or struct. Fields are members of their containing type. Generally, you should use fields only for variables that have private or protected accessibility. Data that your class exposes to client code should be provided through methods, properties, and indexers. By using these constructs for indirect access to internal fields, you can guard against invalid input values. A private field that stores the data exposed by a public property is called a backing store or backing field.

Fields typically store the data that must be accessible to more than one class method and must be stored for longer than the lifetime of any single method.

## Property:

A property is a member that provides a flexible mechanism to read, write, or compute the value of a private field. Properties can be used as if they are public data members, but they are special methods called accessors. This enables data to be accessed easily and still helps promote the safety and flexibility of methods (Encapsulation).

* Properties enable a class to expose a public way of getting and setting values, while hiding implementation or verification code.
* A get property accessor is used to return the property value, and a set property accessor is used to assign a new value. These accessors can have different access levels. For more information, see Restricting Accessor Accessibility.
* The value keyword is used to define the value being assigned by the set accessor.
* Properties can be read-write (they have both a get and a set accessor), read-only (they have a get accessor but no set accessor), or write-only (they have a set accessor, but no get accessor). Write-only properties are rare and are most commonly used to restrict access to sensitive data.
* Simple properties that require no custom accessor code can be implemented either as expression body definitions or as auto-implemented properties.

One basic pattern for implementing a property involves using a private backing field for setting and retrieving the property value. The get accessor returns the value of the private field, and the set accessor may perform some data validation before assigning a value to the private field. Both accessors may also perform some conversion or computation on the data before it is stored or returned.

## Method:

A method consists of:

* Name
* Return type
* Type / kind of its parameters
* Body (block of code)

A standard way to write a method:

public int Add(int a, int b){ code }

At minimum, a method needs:

* An access modifier (the default access modifier is private)
* A return type (void or a datatype)
* An identifier (name)
* Input parameters (can be empty)
* A body that holds the methods code

## Method Overloading:

Method overloading is a form of polymorphism in OOP. Polymorphism allows objects or methods to act in different ways, according to the means in which they are used.

Overloading happens when you have two methods with the same name but different signatures. The compiler will choose the correct method, based on the arguments you input, or throw an exception if something went wrong.

## Constructor:

Whenever a class or struct is created, its constructor method is called. A class or struct may have multiple constructors that take different arguments. Constructors enable the programmer to set default values, limit instantiation, and write code that is flexible and easy to read.

If you do not provide a constructor for your class, C# creates one by default that instantiates the object and sets member variables to the default values.

A constructor must be public and have the exact same name as the class. It can have any amount and type of parameters. This is great since it makes the program safer for a user since you cannot instantiate a class if you do not give the constructor the proper values for the parameters (proper type).

## Constructor Overloading:

Constructor overloading is the exact same as method overloading.

We can overload constructors in different ways as follows:

* By using different type of arguments
* By using different number of arguments
* By using different order of arguments

The compiler will choose the correct constructer, based on the arguments you input, or throw an exception if something went wrong.

## Constructor call to base class:

A derived class can use a base classes constructor with the “base” keyword.

public class Manager : Employee

{

public Manager(int annualSalary)

: base(annualSalary)

{

//Add further instructions here.

}

}

In this example, the constructor for the base class is called before the block for the constructor is executed. The base keyword can be used with or without parameters. Any parameters to the constructor can be used as parameters to base, or as part of an expression.

## Access modifiers:

All types and type members have an accessibility level. The accessibility level controls whether they can be used from other code in your assembly or other assemblies. Use the following access modifiers to specify the accessibility of a type or member when you declare it:

public: The type or member can be accessed by any other code in the same assembly or another assembly that references it.

private: The type or member can be accessed only by code in the same class or struct.

protected: The type or member can be accessed only by code in the same class, or in a class that is derived from that class.

internal: The type or member can be accessed by any code in the same assembly, but not from another assembly.

protected internal: The type or member can be accessed by any code in the assembly in which it is declared, or from within a derived class in another assembly.

private protected: The type or member can be accessed only within its declaring assembly, by code in the same class or in a type that is derived from that class.

The default access modifier is private.

## Return types:

Void: Returns nothing. Usually used for side effects or for printing something to the console.

Any other data type / object.

## Conditionals:

Relational and Equality Operators:

<, >, <=, >=, ==, !=.

If: Do something if the condition is met

Else if: Else do something if the first condition is not met, and this condition is met.

Else: Do something if no condition is met (default case).

Switch: Advanced version of if-else if- else statement.

?: is a short version of an if-else statement:

condition ? consequent : alternative, or a simple version

is this condition true ? yes : no

## Boolean Expressions:

Unary operator requires 1 operand. Binary operator requires 2 operands. Ternary operator requires 3 operands.

And (&&) is true if both expressions are true.

Or (||) is true if at least one expression is true.

XOR (^) is true if exactly one expression is true.

Negation (!) negates the expression.

And and OR short-circuits if the first operand evaluates to: (false for AND, and True for OR).

By combing conditionals and Boolean expressions, you can control the flow of the program.

## Loops:

For: Do something a certain number of times (Initialization, condition, iterator).

Foreach: (Datatype name in some type of list) do something.

While: Do something while a condition is true.

Do-While: Execute code first then do something while a condition is true.

## Exception handling:

An exception is any error condition or unexpected behavior that is encountered by an executing program. Exceptions can be thrown because of a fault in your code or in code that you call (such as a shared library), unavailable operating system resources, unexpected conditions that the runtime encounters (such as code that cannot be verified), and so on. Your application can recover from some of these conditions, but not from others. Although you can recover from most application exceptions, you cannot recover from most runtime exceptions.

An exception is thrown from an area of code where a problem has occurred. The exception is passed up the stack until the application handles it or the program terminates.

You can throw an exception by using the “Throw” keyword. For instance:   
throw new IndexOutOfRangeException();

All exceptions are derived from the “Exception” base class.

A common exception is the “IndexOutOfRangeException” which is thrown when you try to access an item in for instance an array, with an index that is not valid. An example: if you have an array with 10 integers, and try to access the 12th item, then this exception will be thrown.

You can handle exceptions by using a Try-Block.

A Try-Block consists of either:

* Try-Catch
* Try-Finally
* Try-Catch-Finally

A catch block is run if the try block fails to execute properly. You usually either throw an exception or log the error that occurred here. Depending on your code, you can also put your own error handling code in this block.

A finally block is always run no matter what happens in the try block. This can be useful in cases where you need to do something no matter what happens, as for instance: closing a file.

## Generics (<T>):

Generics introduce the concept of type parameters to .NET, which make it possible to design classes and methods that defer the specification of one or more types until the class or method is declared and instantiated by client code. For example, by using a generic type parameter T, you can write a single class that other client code can use without incurring the cost or risk of runtime casts or boxing operations.

Generic classes and methods combine reusability, type safety, and efficiency in a way that their non-generic counterparts cannot.

The T type can be used as the following:

* As the type of a class.
* As the type of a method parameter.
* As the return type of a property.
* As the type of a private member.

A good example of a generic class is the List<T> class.

## Enumerator:

An enumeration type (or enum type) is a value type defined by a set of named constants of the underlying integral numeric type. To define an enumeration type, use the enum keyword and specify the names of enum members. By default, the associated constant values of enum members are of type int; they start with zero and increase by one following the definition text order. You can explicitly specify any other integral numeric type as an underlying type of an enumeration type. You can also explicitly specify the associated constant values

The default value of an enumeration type E is the value produced by expression (E)0, even if zero does not have the corresponding enum member (Default value is the first / top member).

You use an enumeration type to represent a choice from a set of mutually exclusive values or a combination of choices.

## Static class:

A static class is basically the same as a non-static class, but there is one difference: a static class cannot be instantiated. In other words, you cannot use the new operator to create a variable of the class type. Because there is no instance variable, you access the members of a static class by using the class name itself. For example, if you have a static class that is named UtilityClass that has a public static method named MethodA, you call the method as shown in the following example:

UtilityClass.MethodA();

A static class can be used as a convenient container for sets of methods that just operate on input parameters and do not have to get or set any internal instance fields. For example, in the .NET Class Library, the static System.Math class contains methods that perform mathematical operations, without any requirement to store or retrieve data that is unique to a particular instance of the Math class.

## Abstract class:

An abstract class cannot be instantiated. The purpose of an abstract class is to provide a common definition of a base class that multiple derived classes can share. For example, a class library may define an abstract class that is used as a parameter to many of its functions, and require programmers using that library to provide their own implementation of the class by creating a derived class.

Abstract classes may also define abstract methods. This is accomplished by adding the keyword abstract before the return type of the method. For example:

public abstract class A

{

public abstract void MyMethod(parameter1 name);

}

Abstract methods have no implementation, so the method definition is followed by a semicolon instead of a normal method block. Derived classes of the abstract class must implement all abstract methods. When an abstract class inherits a virtual method from a base class, the abstract class can override the virtual method with an abstract method.

Abstract vs virtual method:

Abstract method must be overridden.

A virtual method can be overridden, but only if you want to.

## Inheriting classes:

See also inheritance in principles and patterns.

Inheritance is written as follows:

public class DerivedClass : BaseClass {}

or

public class DerivedClass : Interface1, Interface2… {}

## Overriding:

See also Abstract Class in Programming.

One of the most common methods to override is the Object.ToString method, which can be used to print to the console, or used as text in a GUI etc.

## Library:

See also Static class in Programming.

Is always static and is a collection of methods. Useful for code reusability.

## The Object Class

Supports all classes in the .NET class hierarchy and provides low-level services to derived classes. This is the ultimate base class of all .NET classes; it is the root of the type hierarchy.

## Recursion:

When a method calls itself. Is incredible for extremely specific types of problems. Can be hard to grasp when reading a recursive method. A recursive method can also be “dangerous” to the program if no termination / condition to stop the recursion is there (even though some programming languages has a recursion depth and throws an exception when the depth is reached).

## Interface:

An interface contains definitions for a group of related functionalities that a non-abstract class or a struct must implement. An interface may define static methods, which must have an implementation. Beginning with C# 8.0, an interface may define a default implementation for members. An interface may not declare instance data such as fields, auto-implemented properties, or property-like events.

By using interfaces, you can, for example, include behavior from multiple sources in a class. That capability is important in C# because the language does not support multiple inheritance of classes. In addition, you must use an interface if you want to simulate inheritance for structs, because they cannot inherit from another struct or class.

Interfaces can contain instance methods, properties, events, indexers, or any combination of those four member types. Interfaces may contain static constructors, fields, constants, or operators.

To implement an interface member, the corresponding member of the implementing class must be public, non-static, and have the same name and signature as the interface member.

When a class or struct implements an interface, the class or struct must provide an implementation for all the members that the interface declares but doesn't provide a default implementation for. However, if a base class implements an interface, any class that is derived from the base class inherits that implementation.

You define an interface by using the interface keyword. The name of an interface must be a valid C# identifier name. By convention, interface names begin with a capital I.

## Delegate (single, multi),

A delegate is a type that represents references to methods with a particular parameter list and return type. When you instantiate a delegate, you can associate its instance with any method with a compatible signature and return type. You can invoke (or call) the method through the delegate instance.

Delegates are used to pass methods as arguments to other methods. Event handlers are nothing more than methods that are invoked through delegates. You create a custom method, and a class such as a windows control can call your method when a certain event occurs. The following example shows a delegate declaration:

public delegate int PerformCalculation(int x, int y);

Any method from any accessible class or struct that matches the delegate type can be assigned to the delegate. The method can be either static or an instance method. This makes it possible to programmatically change method calls and plug new code into existing classes.

Delegates have the following properties:

* Delegates allow methods to be passed as parameters.
* Delegates can be used to define callback methods.
* Delegates can be chained together; for example, multiple methods can be called on a single event.
* C# version 2.0 introduced the concept of anonymous methods, which allow code blocks to be passed as parameters in place of a separately defined method. C# 3.0 introduced lambda expressions as a more concise way of writing inline code blocks. Both anonymous methods and lambda expressions (in certain contexts) are compiled to delegate types. Together, these features are now known as anonymous functions.

Types of Delegates:

* Single Cast Delegate
* Multi Cast Delegate

Single Cast Delegate: SingleCast Delegates refer to a single method with matching signature. SingleCast Delegates derive from the System.Delegate class  
  
Multi Cast Delegate: This is a kind of delegates that can refer to multiple methods that have the same signature at one time.

Delegates are great to use for Observer Patterns.

You add methods (if they can be added) to a delegate by using the “+=” operator, and remover them with the “-=” operator.

## Event:

Event is a more specific version of a delegate, No one outside of your class can raise the event.

An event is like a list of method calls. You add methods (if they can be added) to an event by using the “+=” operator, and remover them with the “-=” operator.

Events enable a class or object to notify other classes or objects when something of interest occurs. The class that sends (or raises) the event is called the publisher and the classes that receive (or handle) the event are called subscribers.

Events have the following properties:

* The publisher determines when an event is raised; the subscribers determine what action is taken in response to the event.
* An event can have multiple subscribers. A subscriber can handle multiple events from multiple publishers.
* Events that have no subscribers are never raised.
* Events are typically used to signal user actions such as button clicks or menu selections in graphical user interfaces.
* When an event has multiple subscribers, the event handlers are invoked synchronously when an event is raised. To invoke events asynchronously.
* In the .NET class library, events are based on the EventHandler delegate and the EventArgs base class.

## Event Handler / raising events:

## Anonymous methods:

Anonymous methods provide a technique to pass a code block as a delegate parameter. Anonymous methods are the methods without a name, just the body.

You need not specify the return type in an anonymous method; it is inferred from the return statement inside the method body.

Anonymous methods are declared with the creation of the delegate instance, with a delegate keyword. For example,

delegate void NumberChanger(int n);

...

NumberChanger nc = delegate(int x) {

Console.WriteLine("Anonymous Method: {0}", x);

};

## Func<T,TResult> Delegate:

Encapsulates a method that has one parameter and returns a value of the type specified by the TResult parameter.

|  |  |
| --- | --- |
| 1 | public delegate TResult Func<T1, T2, out TResult>(T1 param1, T2 param2); |

## Action:

Encapsulates a method that has no parameters and does not return a value.

|  |  |
| --- | --- |
| 1 | public delegate void Action<T1, T2>(T1 param1, T2 param2); |

## Predicate:

Represents the method that defines a set of criteria and determines whether the specified object meets those criteria.

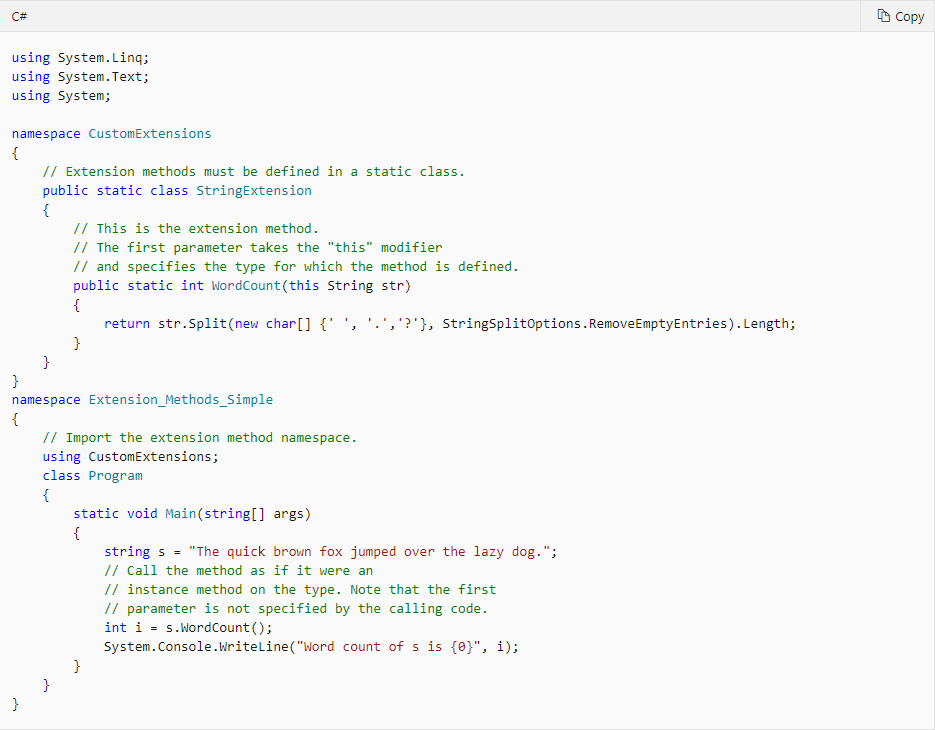
|  |  |
| --- | --- |
| 1 | public delegate bool Predicate<T>(T input); |

## Extension methods:

Extension methods enable you to "add" methods to existing types without creating a new derived type, recompiling, or otherwise modifying the original type. Extension methods are static methods, but they are called as if they were instance methods on the extended type. For client code written in C#, F# and Visual Basic, there is no apparent difference between calling an extension method and the methods defined in a type.

To define and call the extension method

* Define a static class to contain the extension method.
* The class must be visible to client code. For more information about accessibility rules, see Access Modifiers.
* Implement the extension method as a static method with at least the same visibility as the containing class.
* The first parameter of the method specifies the type that the method operates on; it must be preceded with this modifier.
* In the calling code, add a using directive to specify the namespace that contains the extension method class.
* Call the methods as if they were instance methods on the type.
* Note that the first parameter is not specified by calling code because it represents the type on which the operator is being applied, and the compiler already knows the type of your object. You only must provide arguments for parameters 2 through n.

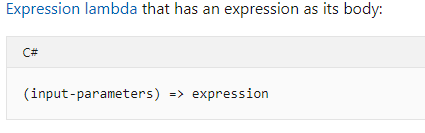
The following example implements an extension method named WordCount in the CustomExtensions.StringExtension class. The method operates on the String class, which is specified as the first method parameter. The CustomExtensions namespace is imported into the application namespace, and the method is called inside the Main method:

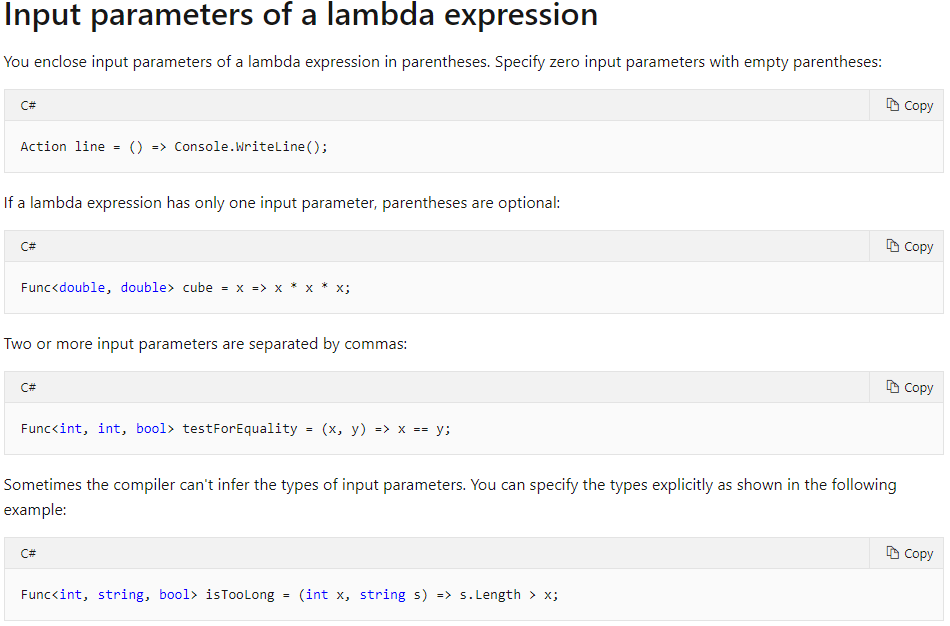
## Lambda:

Use the lambda declaration operator => to separate the lambda's parameter list from its body. To create a lambda expression, you specify input parameters (if any) on the left side of the lambda operator and an expression or a statement block on the other side.

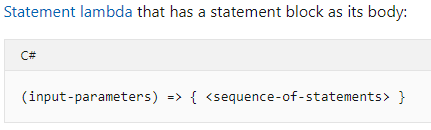
Any lambda expression can be converted to a delegate type. The delegate type to which a lambda expression can be converted is defined by the types of its parameters and return value. If a lambda expression does not return a value, it can be converted to one of the Action delegate types; otherwise, it can be converted to one of the Func delegate types. For example, a lambda expression that has two parameters and returns no value can be converted to an Action<T1,T2> delegate. A lambda expression that has one parameter and returns a value can be converted to a Func<T,TResult> delegate.

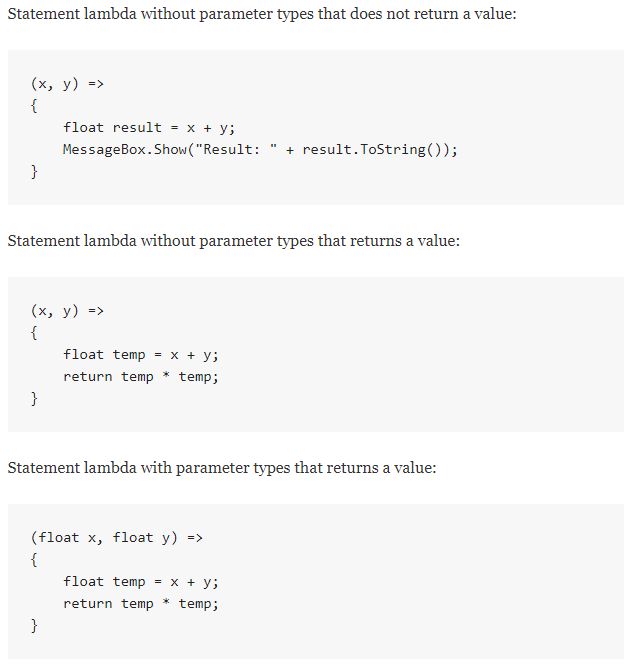
## Lambda expression:





## Lambda statement:





## LINQ (Language Integrated Query):

All LINQ query operations consist of three distinct actions:

* Obtain the data source.
* Create the query.
* Execute the query.

An example is executing the query on a List<T> with a foreach loop:

foreach(var item (from data source) in Query){ do something with item }

## IEnumerable<T> Interface:

Exposes the enumerator, which supports a simple iteration over a collection of a specified type. Basically, all collections implement this interface so that you can iterate over the elements in the collection.

An Enumerable is a class that can give you Enumerators. It has a method called GetEnumerator which gives you an Enumerator that looks at its items. When you write a foreach loop in C#, the code that it generates calls GetEnumerator to create the Enumerator used by the loop.

## IEnumerable Interface:

Is the same as above, but for non-generic types.

## IEnumerator Interface:

Supports a simple iteration over a non-generic collection - the thing that iterates through a type.

## Abstract Data Types:

An ADT (Abstract Data Type) is more of a logical description, while a Data Structure is concrete.

Think of an ADT as a picture of the data and the operations to manipulate and change it.

A Data Structure is the real, concrete thing. It can be implemented and used within an algorithm.

## LinkedList:

A linked list is a sequence of data structures, which are connected via links (references).

Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

* Link − Each link of a linked list can store a data called an element.
* Next − Each link of a linked list contains a link to the next link called Next.
* LinkedList − A Linked List contains the connection link to the first link called First.
* Linked list can be visualized as a chain of nodes, where every node points to the next node.



As per the above illustration, following are the important points to be considered.

Linked List contains a link element called first.

Each link carries a data field(s) and a link field called next.

Each link is linked with its next link using its next link.

Last link carries a link as null to mark the end of the list.

Types of Linked List:

* Simple Linked List − Item navigation is forward only.
* Doubly Linked List − Items can be navigated forward and backward.
* Circular Linked List − Last item contains link of the first element as next and the first element has a link to the last element as previous.

Basic Operations:

* Insertion − Adds an element at the beginning of the list.
* Deletion − Deletes an element at the beginning of the list.
* Display − Displays the complete list.
* Search − Searches an element using the given key.
* Delete − Deletes an element using the given key.

## Data Types:

Bool contains either true or false (1 or 0), which is used in conditionals.

Int contains whole numbers. Default is 0.

Double contains decimal numbers. Default is 0.

Char contains Unicode characters. Default is null.

String is an array of characters. Default is “”.

Array is a collection of a specific datatype, with at fixed length, set at the initialization of the object.

List<T> is a dynamic version of an array, that has a dynamic size. It also features numerous methods that an array does not have, like inserting an element at the end of the list.

# Conversion between data types:

In C#, you can perform the following kinds of conversions:

**Implicit conversions**: No special syntax is required because the conversion always succeeds, and no data will be lost. Examples include conversions from smaller to larger integral types, and conversions from derived classes to base classes.

**Explicit conversions (casts)**: Explicit conversions require a cast expression. Casting is required when information might be lost in the conversion, or when the conversion might not succeed for other reasons. Typical examples include numeric conversion to a type that has less precision or a smaller range, and conversion of a base-class instance to a derived class.

**User-defined conversions**: User-defined conversions are performed by special methods that you can define to enable explicit and implicit conversions between custom types that do not have a base class–derived class relationship.

**Conversions with helper classes**: To convert between non-compatible types, such as integers and System.DateTime objects, or hexadecimal strings and byte arrays, you can use the System.BitConverter class, the System.Convert class, and the Parse methods of the built-in numeric types, such as Int32.Parse.

## Console UI and user Interaction:

You can use the “Console.Readline()” to get a an input from the console.

## Persistence (File I/O):

Absolute path:

The whole path from the drive to the folder with the file you need. For instance, the default folder to save files in Visual Studio: “C:\Users\Sebastian\source\repos”.

Relative:

The path from a given folder to the file you need. For instance, if you set the Visual Studio project folders path, to the base path, then you only need to write the path from the base path to the file you need.

Path.Combine(BasePath, “\tickets.csv”).

Use StreamReader to read a txt file, and StreamWriter to write to a txt file (From System.IO).

Use StreamReader.Readline() to get the next line in the file.

Use StreamWriter.Writeline() to a file.

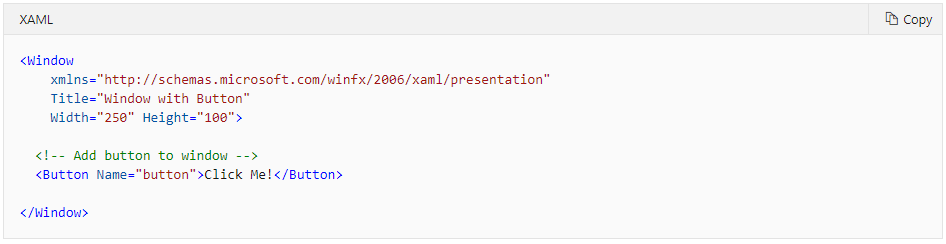
## WPF (Windows Presentation Foundation):

WPF is a built-in project in Visual Studio, and a way to make GUIs. WPF uses buttons textboxes and more to get input from the user. This makes it easier for users to use the program with a GUI, instead of a console application.

A WPF project consist of:

* Markup: XAML that implements the applications appearance and user controls.
* Code behind file:

The following example uses XAML to implement the appearance of a window that contains a single button:



Specifically, this XAML defines a window and a button by using the Window and Button elements, respectively. Each element is configured with attributes, such as the Window element's Title attribute to specify the window's title-bar text. At run time, WPF converts the elements and attributes that are defined in markup to instances of WPF classes. For example, the Window element is converted to an instance of the Window class whose Title property is the value of the Title attribute.

The main behavior of an application is to implement the functionality that responds to user interactions, including handling events (for example, clicking a menu, tool bar, or button) and calling business logic and data access logic in response. In WPF, this behavior is implemented in code that is associated with markup. This type of code is known as code-behind.

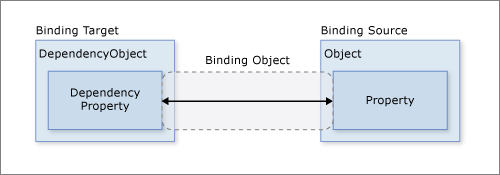
## XAML:

XAML is a declarative markup language. As applied to the .NET Core programming model, XAML simplifies creating a UI for a .NET Core app. You can create visible UI elements in the declarative XAML markup, and then separate the UI definition from the run-time logic by using code-behind files that are joined to the markup through partial class definitions.

## XAML Data Binding:

Data binding is the process that establishes a connection between the app UI and the data it displays. If the binding has the correct settings and the data provides the proper notifications, when the data changes its value, the elements that are bound to the data reflect changes automatically. Data binding can also mean that if an outer representation of the data in an element changes, then the underlying data can be automatically updated to reflect the change. For example, if the user edits the value in a TextBox element, the underlying data value is automatically updated to reflect that change.

Regardless of what element you are binding and the nature of your data source, each binding always follows the model illustrated by the following figure.



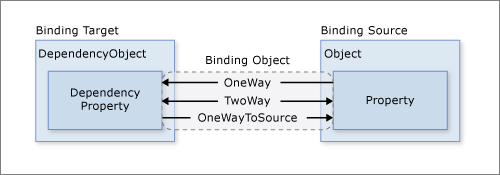
As the figure shows, data binding is essentially the bridge between your binding target and your binding source. The figure demonstrates the following fundamental WPF data binding concepts:

Typically, each binding has four components:

* A binding target object.
* A target property.
* A binding source.
* A path to the value in the binding source to use.

For example, if you want to bind the content of a TextBox to the Employee.Name property, your target object is the TextBox, the target property is the Text property, the value to use is Name, and the source object is the Employee object.

This figure illustrates the different types of data flow:



* OneWay binding causes changes to the source property to automatically update the target property, but changes to the target property are not propagated back to the source property. If there is no need to monitor the changes of the target property, using the OneWay binding mode avoids the overhead of the TwoWay binding mode.
* TwoWay binding causes changes to either the source property or the target property to automatically update the other. This type of binding is appropriate for editable forms or other fully interactive UI scenarios. Most properties default to OneWay binding, but some dependency properties (typically properties of user-editable controls such as the TextBox.Text and CheckBox.IsChecked default to TwoWay binding.
* OneWayToSource is the reverse of OneWay binding; it updates the source property when the target property changes. One example scenario is if you only need to reevaluate the source value from the UI.
* Not illustrated in the figure is OneTime binding, which causes the source property to initialize the target property but does not propagate subsequent changes. If the data context changes or the object in the data context changes, the change is not reflected in the target property. This type of binding is appropriate if either a snapshot of the current state is appropriate, or the data is truly static.

## Command Binding:

## MVVM (Model View Viewmodel):

# Technology:

## Binary representation, OS (program/process, memory, file system):

# Skills:

## Debugging:

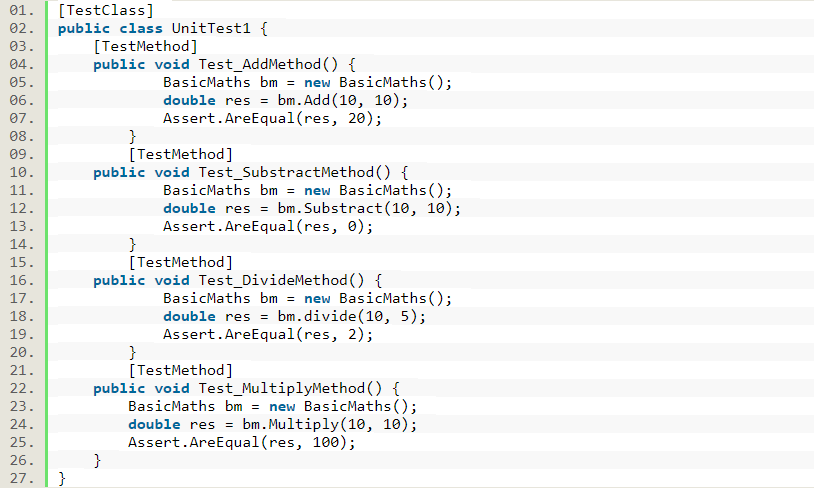
## Testing (Unit-Test):

In the software development process Unit Tests basically test individual parts (also called as Unit) of code (mostly methods) and make it work as expected by programmer. A Unit Test is a code written by any programmer which test small pieces of functionality of big programs. Performing unit tests is always designed to be simple, A "UNIT" in this sense is the smallest component of the large code part that makes sense to test, mainly a method out of many methods of some class. Generally, the tests cases are written in the form of functions that will evaluate and determine whether a returned value after performing Unit Test is equals to the value you were expecting when you wrote the function. The main objective in unit testing is to isolate a unit part of code and validate it is to correctness and reliable.

There are few reasons that can give you a basic understanding of why a developer needs to design and write out test cases to make sure major requirements of a module are being validated during testing,

* Unit testing can increase confidence and certainty in changing and maintaining code in the development process.
* Unit testing always can find problems in early stages in the development cycle.
* Codes are more reusable, reliable, and clean.
* Development becomes faster.
* Easy to automate.

Example on how to write a Unit Test:



## Test Driven Development:

The traditional method of writing unit tests consists of writing the tests to check the validity of the code. First, the code is written, then the tests are written. This is contrary to Test-Driven Development.

Test-Driven Development (TDD) consists of writing the tests before writing the code.

First, the test is written and must fail at the beginning. Then, the code is written so that the test passes. Then, the test must be executed and must succeed. Then, the code is refactored. Then, the test must be executed again to ensure that the code is correct.

To summarize, this is done through five steps:

1. Write a test.
2. The test must fail at the beginning.
3. Write the code so that the test passes.
4. Execute the test and ensure that it passes.
5. Refactor the code.

## Versioning (GIT):

## Naming and coding standards / conventions:

Private = Camelcase.

Public = Pascalcase.

If in doubt just use pascalcase.

## Refactoring:

Shortening of code to make it more concise. You usually only refactor code, after you have made sure that it works. See also Test-Driven Development.

## Redesigning:

## Quality Assuring: