C# revision

SLIDE 1 - LANGUAGE FUNDAMENTALS

PROS OF C# BEING A STRONGLY TYPED OBJECT ORIENTED PROGRAMMING LANGUAGE:

- Easier to read
- Better maintainability
- Better bug finding
- More boilerplate coding

OBJECT ORIENTED PROGRAMMING

- Provides a unified way of working with all variable types
- Gives every object a base set of built in functions

APIE:
Abstraction

Polymorphism
Code that behaves differently depending on the context (can be overloading, the plus-sign, interfaces (shape example), etc.)

Inheritance
Fields, properties, methods, etc.

Encapsulation
Data hiding, accessibility, only exposing the necessary data, e.g. instantiating an object

Implicitly typed variables are when the C# compiler decides the datatype.

Implicit conversion

- Only works when there is no possibility of loss of data.
- The compiler automatically casts it for you.

Explicit conversion

- When there is a danger of losing data
- Telling the compiler to ignore its type safe rules.

Naming conventions

- Variables use Camel case (camelCase)
- Constants, methods and classes use Pascal case (PascalCase)

Enumerations e.g. (enum Test {}) are constants set outside the Main() which will never change.

SLIDE 2 - OPERATORS

Modulus divides the value of one expression by the value of another, and returns the remainder. E.g. 5 % 2 = 1.

Prefix and **postfix** operators determines when a value is incremented. E.g. ++prefix and postfix++.

SLIDE 3 - BRANCHING / LOOPS

Example of parsing a string to an int: **Int32.TryParse("32", out temp);**

SLIDE 4 - COLLECTIONS

Arrays in C# hold a fixed number of elements, all of the same type. int[] array1 = new int[5];

Array functions include myArray.Length(), myArray.Clear(), myArray.Sort() and myArray.Reverse()

Lists are like arrays but can be dynamically resized and can still only hold one data type.

ArrayLists can be dynamically resized and can hold different objects.

LINQ (Language-Integrated Query) allows us to query collections using sql like commands.

```
int[] arrayOfInts = { 20, 2, 3, 4, 5, 6, 7, 8, 9, 20 };
int[] filteredList = (from x in arrayOfInts where x > 7 select x).ToArray();
```

Lambda expressions are anonymous functions. It allows us to write local functions that can be passed as arguments or returned as the value of function calls.

LINQ with Lambda:

```
int[] arrayOfInts = { 20, 2, 3, 4, 5, 6, 7, 8, 9, 20 };
int[] filteredList = arrayOfInts.Where(item => item > 7).ToArray();
```

SLIDE 5 - METHODS

Reasons to use **methods**:
- Better Structured Program and More Readable Code.
- Avoid Duplicated Code
- Code Reuse
- Discrete pieces of functionality code.
- Focus on a single task.
- Be named accordingly after their task.
- The PascalCase rule must be applied.
- Can return a value or not.
- In the C# language, a method can be declared only between the opening "{" and the closing "}" brackets of a class.
- Methods canNOT be declared inside the body of another method.

Contents of methods

```
<access> <return type> <function name>(<parameters>)
{
        <function code>
}
```

Access modifier
Public, private, static plus some others

Return type
Void or any type such as int

Function name
Name that represent the functionality

Parameters
Vars, arrays or objects that is needed for the function to work.

Named Arguments enable us to specify an argument for a particular parameter. Example:

```
static int CalculateBMI(int weight, int height)
  {
     return (weight * 703) / (height * height);
  }
```

The function is called with the values for the parameters with the correct data types.

Optional Arguments enable us to omit arguments for some parameters. If the argument is not given, the parameter will get the default value.

Optional parameters are defined at the end of the parameter list, after any required parameters.

```
public void ExampleMethod(int required, string optionalstr = "default string", int optionalint = 10)
{
   Console.WriteLine("input: {1}, {2}, and {3}.", required, optionalstr, optionalint);
}
```

The method can be called in the normal way.

```
ExampleMethod(1, "One", 1);
ExampleMethod(2, "Two");
ExampleMethod(3);
```

Overloadning is when functions with the same name and different parameter lists are called. Overloading improves clarity, eliminates complexity and enhance performance.

By using the **params** keyword, you can specify a method parameter that takes a variable number of arguments.

```
static void Main()
{
    You can send a comma-separated list of arguments of the specified type.
    UseParams(1, 2, 3, 4);
    An array argument can be passed, as long as the array type matches the parameter type
    int[] myIntArray = { 5, 6, 7, 8, 9 };
    UseParams(myIntArray);
}

static void UseParams(params int[] list)
{
    for (int i = 0; i < list.Length; i++)
    {
        Console.Write(list[i] + " ");
    }
    Console.WriteLine();
}
```

SLIDE 6 - OBJECT-ORIENTED PROGRAMMING

Object orientation is a programming paradigm based on a set of ideas (APIE)

OOP provides structure and makes thinking about programming closer to thinking about the real world, by assigning functionality to certain elements / objects.

Identity: Objects have their own existence, that is independent of other objects.
Characteristics (Attributes): Information that describes their current state. i.e. color, weight or size
Behaviour: Things they can do.

Classes are used to create objects. A class describes what an object will become, but is not the object itself, it is more like a **blueprint**.

The class itself is defined first and hereafter endless objects can be created based on the single class.

When creating a class:
Name (Type):
What is it? A person, event, product, table, car, etc.

Characteristics (Fields, Properties):
What describes it? Width, Height, Color, Size, Score, FileType

Behaviour (Methods):
What can it do? Print, Play, Open, Close, Save.

Class Members
Fields and properties represent information that an object contains.
Fields are like variables because they can be read or set directly.
Properties have **get and set** procedures, which provide more control on how values are set or returned.

Methods are actions that an object can perform

Constructors are class methods that are executed automatically when an object of a given type is created. A constructor can run only once when a class is created.

Constructors are automatically called when an object is created. They do not return values and their purpose is to initiate the state of an object. Constructors can be overloaded.

Properties are used when you might want an external class to read (get) but not change (set).
You can make the field only accept values in a certain range, have different access modifiers for the same field and is an example of encapsulation.

ABSTRACTION
Abstraction focuses on the essential elements of something rather than the specifics. Everything irrelevant will be discarded. This means that one single class will be created for e.g. a bank account with its specific characteristics and behaviour. The class will be a blueprint which is suited for all people with a bank account, rather than having a bank account class for everyone.

POLYMORPHISM

SLIDE 8 - POLYMORPHISM & INTERFACES

Polymorphism is the ability for an object to change its public interface according to how it's being used.

The **is** operator can be used to test if an object is of a specific type if(shape is triangle){}

An **interface** is a code structure that is similar to an abstract class that has no concrete members. An interface can contain public members such as properties and methods but they can't have any functionality. An interface can be used to define what a class must do, but not how it will achieve it. Eg. IPrintable, IMailable, etc.

Classes tend to be things, whereas interfaces tend to be behaviours.

Defining an interface:

```
interface IPrey
{
   int FleeSpeed { get; set; }
   void Flee();
}

class Fish : IPrey
{
    public int FleeSpeed { get; set;}

    public void Flee()
    {
        Console.WriteLine("Fish fleeing");
    }
}
```

INHERITANCE

SLIDE 7 - INHERITANCE

Inheritance is the hierarchical relationships among the classes. By declaring that a class inherits from an existing class, the elements of the old class will be added to the new one. The specialised class is called the **derived** class. The more generalised class is called the **base** class. Specialised classes can only inherit from one class.

The **protected** access modifier is used within inheritance hierarchies. A protected class is as visible as a private class but if an item in a base class is declared protected, it is visible in the sub classes.

Override is when a derived class modifies the behaviour of a super class' methods and properties. E.g. if a parent class has a public virtual void method which outputs for example a string, this output can be overridden in the derived class.

```
class Parent
{
    public virtual void SaySomething()
    {
        Console.WriteLine("Don't do that");
    }
```

```
}

class Child:Parent
{
    public override void SaySomething()
    {
        Console.WriteLine("Let's do it!");
    }
}
```

Base methods allows sub classes to access base classes' functionalities.

Abstract classes allows you to create classes and class members which must be implemented in derived classes. If a class contains at least one abstract method, the class itself will have to be declared abstract. In the derived classes, the method public **override** void must be implemented.

Preventing inheritance can be done with the **sealed** keyword which is useful for classes which contain security functionality.

ENCAPSULATION
The idea of encapsulation is to encapsulate the objects' attributes and methods and hide everything about that object except what is necessary (when methods' access modifiers are public, protected, private).

SLIDE 9 - ASP.NET MVC BASICS

MVC is a way of structuring web applications by defining the app with 3 layers:

THE MODEL contains or represents the data that users work with, and rules for manipulating that data.

A **strongly typed** view is a view that binds with a model. This allows you to access model properties in that view and get intellisense.

Model binding is when using all the data coming from HTTP and use the DefaultModelBinder to do all the type conversion and mapping of the values to the model properties. This means using properties from a model in a controller.

Html.EditorFor allows you to retain control over the display of form elements by data type.

```
@Html.EditorFor(model => model.Text, new { htmlAttributes = new { @class = "form-control" } })

//Displays model property name
@Html.DisplayNameFor(model => model.Title)

//Displays model property value
@Html.DisplayFor(model => model.Title)

//Displays property validation message
@Html.ValidationMessageFor(model => model.Title,"", new{ @class = "text-danger" })
```

Validation can be the keyword [Required] which means an input field must be filled out. It can be used with e.g. minimum length of password, provide error messages, etc.

```
public class Post
{
    [Required (ErrorMessage="Come on you douche")]//Required with custom error message
    public string Title { get; set; }
    [Required]
    [StringLength(160, MinimumLength = 3)] //Requires length between 3 and 160
    public string Body { get; set; }
}
```

ModelState has two purposes: to store the value submitted to the server, and to store the validation errors associated with those values.

THE VIEW is the parts of the application that handles the display of the data. Most often the views are created with the data from the model.

We use view or template files to generate HTML responses to the client. The view file is created using the Razor view engine.

The **viewbag** is a property of the controller. It is used as a dynamic object that can store properties.

Razor is a markup syntax that lets you embed server base code (C#) into web pages. Razor is initialised with a @.

HTML helpers are methods invoked on the HTML property of a view.

The **Html.ActionLink** does not link to a view, it links to a controller action which updates automatically based on route.config. Example with Razor: @Html.ActionLink("About this Website", “About") which would output the html About this Website

The **Html.Action** helper lets you call controller actions that will return data as an html string.

Layouts in **Razor** helps maintain a consistent look. Layout can be used to define a common template for the design. @RenderBody can be used to specify the location where views using this layout will have their main content rendered.

Html.Partial renders the partial view as an HTML encoded string. The method can be stored in a variable.

Html.BeginForm() is used for easily creating forms. If no arguments are specified, the Html.BeginForm() will create a POST form that points to the current controller and current action.

```
[HttpPost]
public ActionResult Create(string username, string password)
{
    //Create user
}
```

THE CONTROLLER is the part of the application that handles user interaction (http flow). Typically controllers receive data from a view (get, post), and sends input data to the model.

The default controller routing is /[Controller]/[ActionName]/[Parameters]. The first part of the URL determines the **controller** class to execute. The second part determines the **action** method to be executed. The third part are **parameters**.

The parameters are for route data. By default the third URL parameter matches the route parameter ID. If a controller action method contains a parameter that matches what is specified in the URL, the value will be passed to that.

SLIDE 10 - ENTITY FRAMEWORK

Entity Framework allows you to omit access plumbing code. It works as an ORM used for binding model and database tables together and makes the data storage object oriented.

Entity Framework will default to using **LocalDB** which is a light version of SQL.

The database **context** class is a facade for the database itself which will handle fetching, storing and updating class instances in the database. The database Context derives from the DbContext base class. DbContext is responsible for:

- **EntitySet**: DbContext contains entity set (DbSet<TEntity>) for all the entities which is mapped to DB tables.
- **Querying**: DbContext converts LINQ-to-Entities queries to SQL query and send it to the database.
- **Change Tracking**: It keeps track of changes occurred in the entities after it has been querying from the database.
- **Persisting Data**: It also performs the Insert, update and delete operations to the database, based on the entity states.
- **Caching**: DbContext does first level caching by default. It stores the entities which has been retrieved during the life time of a context class.
- **Manage Relationship**: DbContext also manage relationship using CSDL, MSL and SSDL in DB-First or Model-First approach or using fluent API in Code-First approach.
- **Object Materialisation**: DbContext converts raw table data into entity objects.

The **code first** approach let's you code first and create the database structure later, rather than having the database structure set first.

The primary key default convention is that code-first would create a primary key for a property if the property name is Id or <class name>Id.

In order to add the **Database Context class** you need to implement System.Data.Entity in the beginning of the file.

The default behaviour of Entity is to create a database only if it doesn't exist (and throw an exception if the model has changed and the database already exists). In the **initializer** class you'll specify that the database should be dropped and re-created whenever the model changes.

```
public class DbInitializer :DropCreateDatabaseIfModelChanges<AppContext>
{

}
```

Dropping the database causes the loss of all your data. This is generally OK during development, because the **Seed** method will run when the database is re-created and will re-create your test data.

A **ViewModel** is a model that only exists to supply information for a view. It contains the fields which are represented in the strongly typed view and is used to pass data from controller to strongly typed view.

Lazy loading means delaying the loading of related data until it is requested and should be defined as public virtual.

Eager loading is the process whereby a query for one type of entity also loads related entities as part of the query. Eager loading is achieved using the Include method of IQueryable.

SLIDE 11 - JAVASCRIPT IN MVC

Javascript in asp.net mvc should be stored in static files in the root folder 'Scripts'.
Js-files should then be referenced in the view through the scripts section, and this can easily be done by dragging them there.

```
@section scripts
{
<script src="~/Scripts/my.js"></script>
}
```

Unobtrusive validation is a client side validation, that automatically validates input based on the model data annotation using data attributes (data-val, data-val-required, etc.)

A derived class of ActionResult is **JsonResult**. With this we can easily turn c# object into Json.

```
//Example with anonymous object
return Json(new { id = 2, name = "Rune"}, JsonRequestBehavior.AllowGet);
//Example with model
Post post = new Post{name = "Rune", password="password"}
return Json(post, JsonRequestBehavior.AllowGet);
```

SLIDE 12 - SIGNALR

SignalR is an open source .NET library enabling real-time broadcasts of data on the web. It's ideal for any application that needs to send data to clients in real time without requiring an initial request to the server.

SLIDE 12 - IDENTITY

An example of identity is marking an action method with the [Authorize] attribute.