MONASH University

FIT5032 - Internet Applications Development

Introduction to C#

Prepared by - Jian Liew

Last Updated - 31st July 2018

Monash University (Caulfield Campus)

MONASH University

Outline

- Introduction to the .NET
- Introduction to C#
- What is Component Oriented Programming?
- Overview of the C# language
- Introduction to Language Integrated Query (LINQ)

MONASH University

What is .NET?

Before we can explain what C# is there is a need to understand what .NET is;

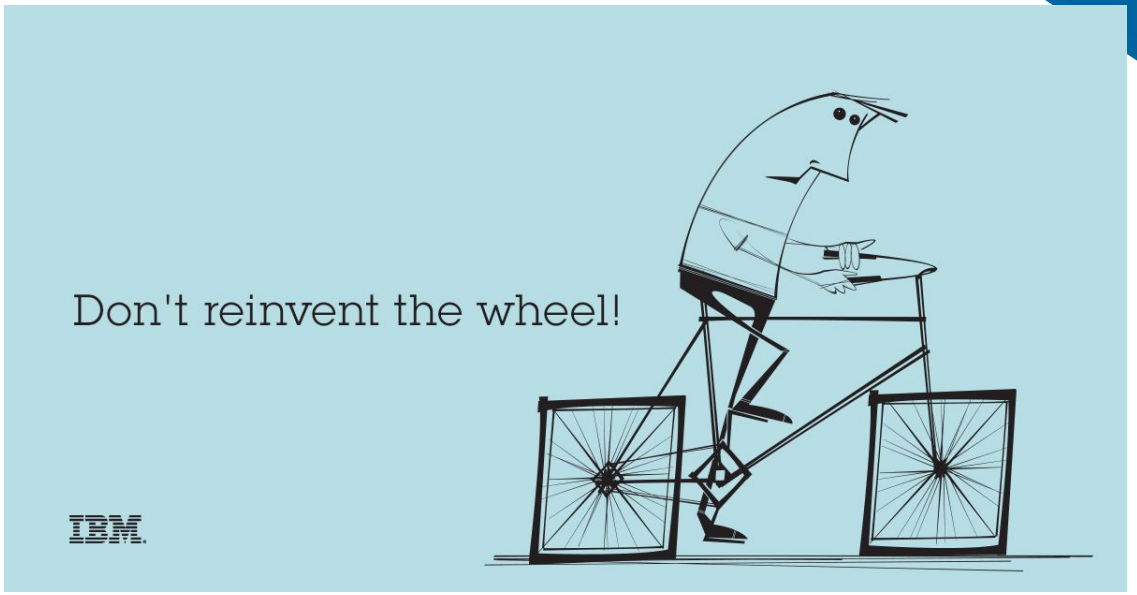
- .NET is a ***free***, ***cross-platform***, ***open source*** developer platform for building many different types of applications.
- With .NET, you can use multiple languages, editors, and libraries to build for web, mobile, desktop, gaming and IoT.
- .NET apps can be written in C#, F# or Visual Basic (VB).
- C# is a **simple**, **modern**, **object-oriented**, and **type-safe** programming language.
- We will not be covering F# or Visual Basic as our main focus is C#.
- **C# is part of the .NET ecosystem.**

.NET **Core** is cross platform while .NET **Framework** is not. It is important to know there is a difference.

It was really big news when .NET became open source. This happened in the year 2014. (4 years ago)

Amount of programming needed

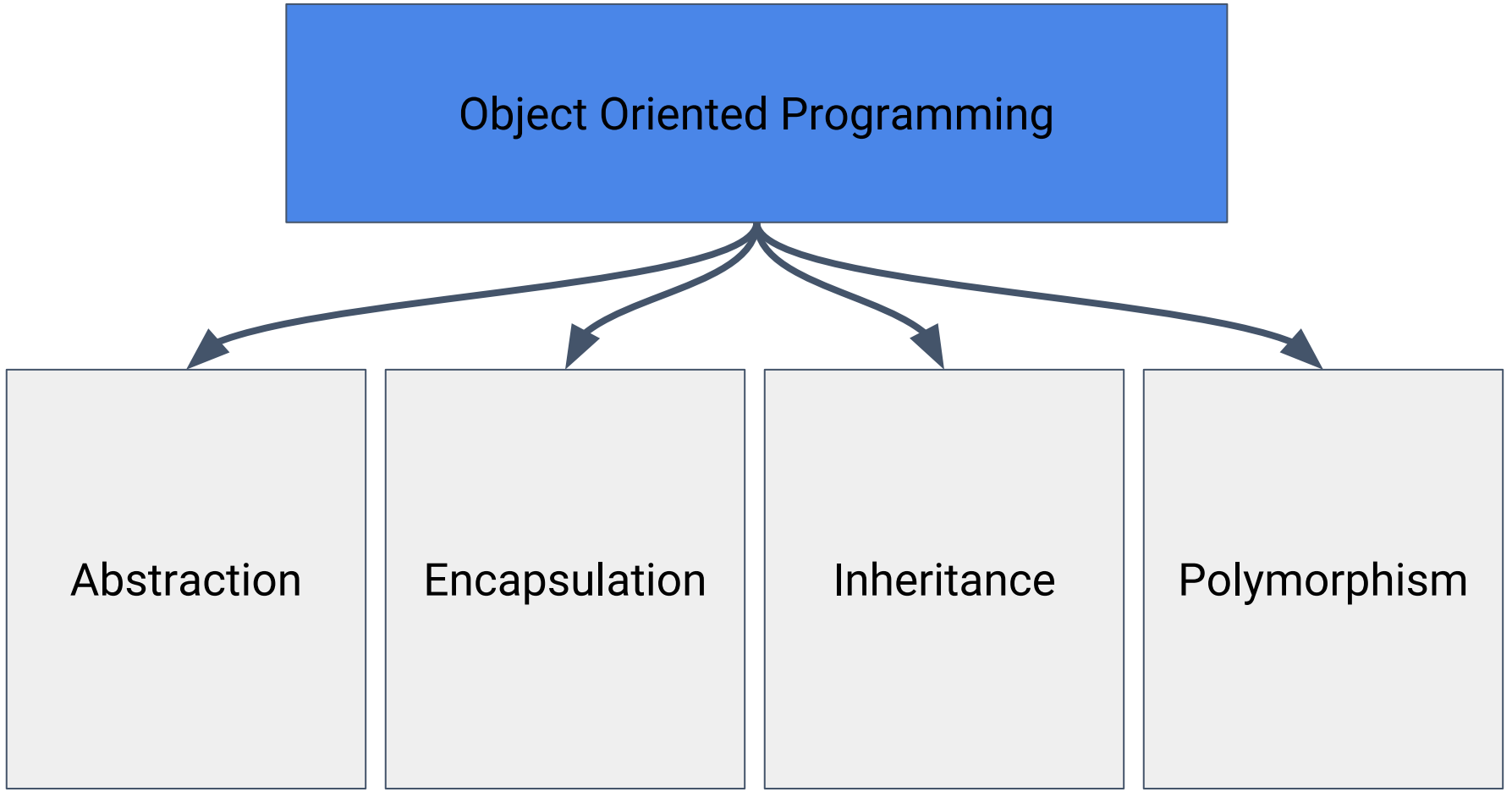
- The amount of programming needed for this subject is actually **considerably smaller in comparison to other subjects.**
- In this subject, we will be using tools which are ready made for us to simplify the development process, but it is still important for us to understand how to properly utilise the tools.
- In other words, we will try “not reinvent the wheel”.
- However, at the end of the day everything depends on the context. This subject is designed so that we try to avoid “reinventing the wheel”. In real life, however often times there are times when reinventing the wheel is needed.
- That being said, it is important to understand the basics of what we will be using.



What is C#?

- C# is pronounced (“See Sharp”). Please do not call it “See HashTag” or “See Pound”.
- It is a **simple**, **modern**, **object-oriented**, and **type-safe programming.**
- C# supports **component-oriented programming.**
- C# has a **unified type system**. This means that all C# types, including “primitive types” such as int and double inherit from a single root object. So, they all share a set of common operations and values of any type can store, transported, and operated upon in a consistent manner.
- The type-safe nature of the language makes it **impossible** to read from uninitialized variables, to index arrays beyond their bounds, or to perform unchecked type cast.

Four Pillars of Object Oriented Programming



In fact, some literature would suggest that there are only 3 pillars.

If you have completed a Foundation level unit, you should understand these basic concepts.

What is Component Oriented Programming?

- C# is a language that supports **Component Oriented Programming**
- It can be said that C# is an object-oriented language that lends itself to be **easily packaged into components.**
- Component-oriented programming is a technique of developing software applications by **combining pre-existing and new components**, much the same way automobiles are built from other components.
- These software components are self-contained, self-describing packages of functionality containing definitions of types that expose both behavior and data.
- The idea of component oriented programming is to create reusable codes in the form of components that can be interchanged.
- Component oriented programming heavily relies on: polymorphism, encapsulation, late binding, inheritance (through interfaces) and most importantly **binary re-usability**. (achieved with .dlls)

Overview of the C# language

The following slides will provide an overview of the C# language.

These overviews will provide basic information about all elements of the language and give you the information necessary to dive deeper into elements of the C# language.

- Program structure
- Types and variables
- Overview of the C# Type System
- Expressions
- Statements
- Classes and objects
- Structs
- Arrays
- Interfaces
- Enums
- Delegates

The following slides are the introduction to the basics of the C# language.

It should be familiar to you if you have done a foundation level unit. There some slightly newer concepts like **“Interfaces”, “Enums” and “Delegates”** that might be new to you.

Program Structure

- Key concepts in C# are **programs, namespaces, types, members**, and **assemblies.**
- C# programs consist of one or more source files.
- Programs declare types, which contain members and can be organized into namespaces.
- Classes and interfaces are examples of types.
- Fields, methods, properties, and events are examples of members.
- When C# programs are compiled, they are physically packaged into assemblies.
- These assemblies have the extension .exe or .dll.

Types and variables

- There are two kind of types in C#: **value types** and **reference types.**
- Variables of value type directly contain their data.
- Reference types store references to their data, this being known as objects.
- Value types can be broken down into **simple types, enum types, struct types** and **nullable value** types.
- Reference types are further divided into class types, interface types, array types, and delegate types.
- It is important to notice that C#, uses **value type** and **not primitive unlike Java.** Officially, **primitive types are not defined in the C# language specification.**

Notice that we have “nullable” value type. What do you think this is? Java 8 introduced the concept of Optional, what do you think is the difference of it and nullable value type?

Overview of C# Type System

- Value types
  - Simple Types
    - Signed integral: `sbyte, short, int, long`
    - Unsigned integral: byte, ushort, uint, ulong
    - Unicode characters: char
    - IEEE floating point: float, double
    - High-precision decimal: decimal
    - Boolean: bool
  - Enum
  - Struct
  - Nullable value types

- Reference Types
  - Class types
    - Ultimate base class of all other types: object
    - Unicode strings: string
    - User-defined types of the form class C {...}
  - Interface types
    - User-defined types of the form interface I {...}
  - Array types
    - Single- and multi-dimensional, for example, int[] and int[,]
  - Delegate types

Notice how it is not called “Primitive Type” but **Value Type.**

Expressions

- Expressions are constructed from **operands** and **operators**.
- The operators of an expression indicate which **operations to apply to the operands**.
- Examples of operators include +, -, *, /, and new.
- Examples of operands include **literals, fields, local variables,** and **expressions.**
- When an expression contains multiple operators, the **precedence of the operators** controls the order in which the individual operators are evaluated.
- Most operators can be overloaded. **Operator overloading** permits user-defined operator implementations to be specified for operations where one or both of the operands are of a user-defined class or struct type.
- **Java does not have operator overloading.** Please note that operator overloading is not method overloading.
- Operator overloading **gives you an opportunity to fully integrate your data type into the language,** so that it can behave as if it were a built-in data type. This can make the code that uses your class more concise and (hopefully) easier to read.

Statements

- The actions of a program are expressed using statements.
- C# supports several different kinds of statements, a number of which are defined in terms of embedded statements.
- A block permits multiple statements to be written in contexts where a single statement is allowed. A block consists of a list of statements written between the delimiters **{ and }**.
- **Declaration statements** are used to declare local variables and constants.
- **Expression statements** are used to evaluate expressions.
- **Selection statements** are used to select one of a number of possible statements for execution based on the value of some expression. In this group are the if and switch statements.
- **Iteration statements** are used to execute repeatedly an embedded statement. In this group are the **while, do, for, and foreach** statements.
- **Jump statements** are used to transfer control. In this group are the **break, continue, goto, throw, return, and yield** statements.
- You can view examples here.

Classes and objects

- Classes are the most fundamental of C#’s types.
- A class is a data structure that **combines state (fields)** and **actions (methods and other function members)** in a single unit.
- A class provides a definition for dynamically created instances of the class, also known as objects.
- Classes support **inheritance and polymorphism**, mechanisms whereby derived classes can extend specialise base classes.
- New classes are created using class declarations.
- Instances of classes are created using the new operator, which allocates memory for a new instance, invokes a constructor to initialize the instance, and returns a reference to the instance.

Structs

- Like classes, structs are **data structures** that can contain data members and function members, but unlike classes, structs are value types and **do not require heap allocation**.
- A variable of a struct type directly stores the data of the struct, whereas a variable of a class type stores a reference to a dynamically allocated object.
- Struct types do not support user-specified inheritance, and all struct types implicitly inherit from type **ValueType**, which in turn implicitly inherits from object.
- Structs are particularly useful for small data structures that have value semantics.
- Complex numbers, points in a coordinate system, or key-value pairs in a dictionary are all good examples of structs.
- The use of structs rather than classes for small data structures can make a large difference in the number of memory allocations an application performs.

Arrays

- An array is a **data structure** that contains a number of variables that are accessed through **computed indices**.
- The variables contained in an array, also called the elements of the array, are all of the same type, and this type is called the element type of the array.
- Array types are **reference types**, and the declaration of an array variable simply sets aside space for a reference to an array instance.
- Actual array instances are created dynamically at runtime using the new operator.
- The new operation specifies the length of the new array instance, which is then fixed for the lifetime of the instance.
- The indices of the elements of an array range from 0 to Length - 1.
- So, C# is a **0 index based language.** Java is another example of a 0 index based language.
- **Can you think of a language that is not a 0 based index? There is a high chance nobody in this class worked with a 0 based index language before.**

Interfaces

- An interface defines a contract that can be implemented by classes and structs.
- An interface can contain methods, properties, events, and indexers.
- An interface **does not provide implementations of the members it defines**—it merely specifies the members that must be supplied by classes or structs that implement the interface.
- Interfaces may employ multiple inheritance.
- By using interfaces, you can, for example, include behavior from multiple sources in a class.
- The notion of **programming to an interface** is a very important concept that developer should know.
- If you are interested please visit this link here.

"Programming to an interface" is an important concept if you wish to be a developer.

Enums

Enums are an important concept, however there are times there are drawbacks.

- An enum type is a **distinct value type with a set of named constants**.
- Enums use one of the integral value types as their underlying storage. They provide semantic meaning to the discrete values.
- An enumeration type or enum provides an efficient way to define a set of named integral constants that may be assigned to a variable
- For example, assume that you have to define a variable whose value will represent a day of the week. There are only seven meaningful values which that variable will ever store.
- **Advantages** of using an enum instead of a numeric type
  - Reduces errors caused by transposing or mistyping numbers.
  - Makes it easy to change values in the future.
  - Makes code easier to read, which means it is less likely that errors will creep into it.
  - Ensures forward compatibility. With enumerations, your code is less likely to fail if in the future someone changes the values corresponding to the member names
- **Developers should always know when to use enums.**

Attributes

- Attributes provide a powerful method of associating metadata, or declarative information, with code (assemblies, types, methods, properties, and so forth).
- After an attribute is associated with a program entity, the attribute can be queried at run time by using a technique called reflection.
- An attribute is a **declarative tag** that is used to convey information to runtime about the behaviors of various elements like classes, methods, structures, enumerators, assemblies etc.
- Please note that these are different in comparison to attributes in Java. **(They are known as annotations in Java where you have the @tag)**
- More information can be found here.

Attributes are a very interesting feature of the C# language. However, a majority of people **would incorrectly refer it as annotations** due to the popularity of Java which coined the term. There is nothing wrong though.

Asynchronous Programming

- **Avoid performance bottlenecks and enhance the overall responsiveness** of your application by using asynchronous programming.
- Asynchrony is essential for activities that are potentially blocking, such as web access. (For example when requesting for JSON data)
- Access to a web resource sometimes is slow or delayed.
- If such an activity is blocked in a synchronous process, the entire application must wait.
- In an asynchronous process, the application can continue with other work that doesn't depend on the web resource until the potentially blocking task finishes.
- So, making I/O calls asynchronous means that your thread does not need to block, and can be freed to do other work.
- **This is a great impact on user experience.**

It is very important to understand when to use an Async task. For example, using **Web APIs these days is becoming more and more common**, so when doing a large request it should be done asynchronously.

Collections

- Collections provide a **more flexible way** to work with groups of objects.
- Unlike arrays, the group of objects you work with can grow and shrink dynamically as the needs of the application change.
- The following table lists some of the frequently used classes in the System.Collections namespace:
  - Dictionary<TKey,TValue> → a collection of key/value pairs that are organized based on the key. **(This is very similar to a HashMap in Java)**
  - ArrayList → an array of objects whose size is dynamically increased as required.
  - Hashtable → a collection of key/value pairs that are organized based on the hash code of the key. **(This is different compared to Dictionary)**
  - Queue → a first in, first out (FIFO) collection of objects.
  - Stack → a last in, first out (LIFO) collection of objects.

It is important to know the difference between each collection type and how they impact the values which they store and their performance.

C# Coding Convention

- The coding convention guide can be found here.
- Coding conventions serve the following purposes:
  - They create a consistent look to the code, so that readers can focus on content, not layout.
  - They enable readers to understand the code more quickly by making assumptions based on previous experience.
  - They facilitate copying, changing, and maintaining the code.
  - They demonstrate C# best practices.
- There are two appropriate ways to capitalize identifiers, depending on the use of the identifier:
  - **PascalCasing → used for all identifiers except parameter names**
  - camelCasing → used only for parameter names.

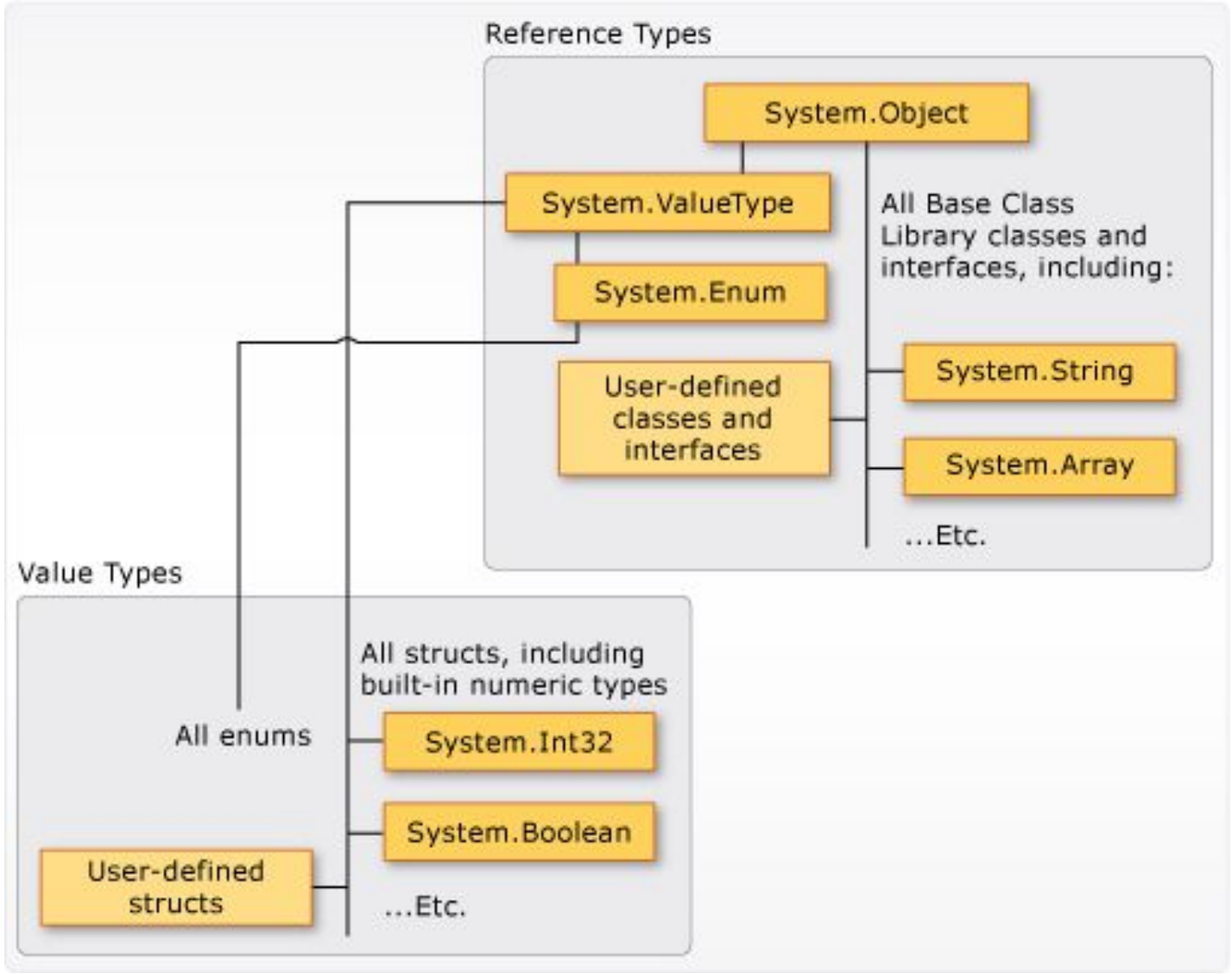
This is different compared to what you are used to.

Strings

- A string is an object of type String whose value is text.
- Internally, the text is stored as a **sequential read-only collection of Char objects. (Think of it as a sequence of characters)**
- There is no null-terminating character at the end of a C# string; therefore a C# string can contain any number of embedded null characters ('\0').
- In C#, the **string keyword is an alias for String.**
- Therefore, String and string are equivalent, and you can use whichever naming convention you prefer.
- In addition, the C# language overloads some operators to simplify common string operations. **(Hence you can use the + operator to concatenate Strings)**
- String objects are immutable: they cannot be changed after they have been created.
- String operations in .NET are highly optimized and in most cases do not significantly impact performance.

Types

- C# is a strongly-typed language.
- Every variable and constant has a type, as does every expression that evaluates to a value.
- Every method signature specifies a type for each input parameter and for the return value.
- A typical C# program uses types from the class library as well as user-defined types that model the concepts that are specific to the program's problem domain.
- It is important to understand two fundamental points about the type system in .NET:
  - Each type in the CTS is defined as either a value type or a reference type.
  - It supports the principle of inheritance. Types can derive from other types, called base types.
- All types, including built-in numeric types such as System.Int32 (C# keyword: int), derived ultimately from a single base type, which is System.Object (C# keyword: object).
- This **unified type hierarchy is called the Common Type System (CTS)**.



- The Unified Type hierarchy.
- Notice that **everything inherits from System.Object**

Nullable

- A nullable type can represent the correct range of values for its **underlying value type**, plus an additional null value.
- For example, a `Nullable<Int32>`, pronounced "Nullable of Int32," can be assigned any value from `-2147483648` to `2147483647`, or it can be assigned the null value.
- A `Nullable<bool>` can be assigned the values `true`, `false`, or `null`.
- The ability to assign null to numeric and Boolean types is especially **useful when you are dealing with databases** and other data types that contain elements that may not be assigned a value.
- More information can be found here.
- If you are interested, you might wonder what is the difference between the Nullable in C# in comparison the the Optional in Java. Link can be found here.

Implicit Typed Local Variables

- Local variables can be declared ***without giving an explicit type***.
- The var keyword instructs the compiler to **infer the type of the variable** from the expression on the right side of the initialization statement.
- The inferred type may be a built-in type, an anonymous type, a user-defined type or a type defined in the .NET Framework class library.

```
// i is compiled as an int
var i = 5;
// s is compiled as a string
var s = "Hello";
// a is compiled as int[]
var a = new[] { 0, 1, 2 };
// expr is compiled as IEnumerable<Customer>
// or perhaps IQueryable<Customer>
var expr =
    from c in customers
    where c.City == "London"
    select c;
// anon is compiled as an anonymous type
var anon = new { Name = "Terry", Age = 34 };
// list is compiled as List<int>
var list = new List<int>();
```

- The var keyword **does not mean "variant"** and **does not indicate that the variable is loosely typed, or late bound**.
- It just means that the compiler determines and assigns the most appropriate type.
- However, the usage of it have the potential of making code harder to understand for other developers, so it is only recommended to use it when it is required.

In other news, Java 10 recently introduced local variable type inference (This is in the JEP 286 specification).

Classes and Structs

- Classes and structs are two of the basic constructs of the common type system in the .NET Framework.
- Each is essentially a **data structure** that **encapsulates a set of data and behaviors that belong together as a logical unit.**
- A class is a **reference type**. When an object of the class is created, the variable to which the object is assigned holds only a reference to that memory.
- A struct is a **value type**. When a struct is created, the variable to which the struct is assigned holds the struct's actual data. When the struct is assigned to a new variable, it is copied.
- In general, **classes are used to model more complex behavior**, or data that is intended to be modified after a class object is created. Structs are best suited for **small data structures that contain primarily data** that is not intended to be modified after the struct is created.

Class

- A class is a construct that enables you to create your own custom types by grouping together variables of other types, methods and events.
- A class is like a blueprint.
- It defines the data and behavior of a type.
- A class defines a type of object, but it is not an object itself. (People often use this interchangeably however they are not the same.)
- An object is a concrete entity based on a class, and is sometimes referred to as an instance of a class.
- In C# can only directly inherit from one base class.
- A class can be declared abstract.
- An abstract class contains abstract methods that have a signature definition but no implementation.
- **Abstract classes cannot be instantiated. They can only be used through derived classes that implement the abstract methods.**

Inheritance

- 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** that is 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.
- 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.

Polymorphism

- Polymorphism is often referred to as the third pillar of object-oriented programming, a**fter encapsulation and inheritance.**
- Polymorphism is a Greek word that means "many-shaped" and it has two distinct aspects:
  - At run time, objects of a derived class may be treated as objects of a base class in places such as method parameters and collections or arrays. When this occurs, the object's declared type is no longer identical to its run-time type.
  - Base classes may define and implement virtual methods, and derived classes can override them, which means they provide their own definition and implementation.
- Virtual methods enable you to work with groups of related objects in a uniform way.
- The best real life example is of an Air Conditioner (AC). **Is is a single device that performs different actions according to different circumstances.** During summer it cools the air and during winters it heats the air.

Constants

- Constants are immutable values which are known at compile time and do not change for the life of the program.
- Constants are declared with the const modifier.
- C# does not support const methods, properties, or events.
- The constants refer to fixed values that the program may not alter during its execution.
- Constants are important because
  - Constants provide **some level of guarantee that code can't change the underlying value.** This is not of much importance for a smaller project, but matters on a larger project with multiple components written by multiple authors.
  - Constants also provide **a strong hint to the compiler for optimization**. Since the compiler knows the value can't change, it doesn't need to load the value from memory and can optimize the code to work for only the exact value of the constant.

Properties

- Properties can be seen as an extension of fields and are accessed using the same syntax.
- They use accessors through which the values of the private fields can be read, written or manipulated.
- A property is a member that provides a flexible mechanism to read, write, or compute the value of a private field.
- In C#, the **auto-property feature** allows property-declaration **to be more concise** when no additional logic is required in the property accessors.
- For example, instead of writing the entire setMethod's body, in C# this can be **accomplished by just specifying the get and set at the field itself.**
- This is not possible in Java as Java does not have this feature.

```
2 references | 0 exceptions
public string LoginProvider { get; set; }
2 references | 0 exceptions
public string RedirectUri { get; set; }
3 references | 0 exceptions
public string UserId { get; set; }
```

The auto property feature allows you to do things like that for the set and get methods. Notice how they are actually public instead of private.

Lambda Expressions

- A lambda expression is **an anonymous function** that you can use to create delegates or expression tree types.
- By using lambda expressions, you **can write local functions that can be passed as arguments** or **returned as the value of function calls**.
- Lambda expressions are particularly helpful for writing LINQ query expressions. (LINQ will be explained on the next slide)
- In short, lambda expressions **provides a way to quickly write functions without naming them.**

Delegates

- This is an important feature of C#
- A delegate is a type that represents references to methods with a particular parameter list and return type.
- **Delegates make it possible to treat methods as entities that can be assigned to variables and passed as parameters.**
- Delegates are similar to the concept of function pointers found in some other languages, but unlike function pointers, delegates are object-oriented and type-safe.
- An interesting and useful property of a delegate is that it does not know or care about the class of the method it references; all that matters is that the referenced method has the same parameters and return type as the delegate.
- In a way because of Delegates, C# **can achieve functional programming.**
- Please see here for more a more descriptive explanation.

Language Integrated Query (LINQ)

- Innovation introduced in the .NET Framework version 3.5
- The aim is to **bridge the gap** between the world of objects and the world of data.
- LINQ makes a query first-class language construct in C#.
- A ***query*** is an expression that retrieves data from a data source. Queries are usually expressed in a specialised query language.
- Different languages have been developed over time for various types of data sources, for example SQL for relational databases and XQuery for XML.
- Developers traditional **need to learn a new query** language for each type of the data source or data format that they must support.
- **LINQ simplifies this situation by offering a consistent model for working with data across various kinds of data sources and formats.**
- In LINQ, you are always working with objects.

To access information stored in a database, we normally write SQL queries, but with LINQ, we do not need to.

Three Parts of a Query Operation

All LINQ query operations consist of three distinct actions

1. Obtain the data source
2. Create the query
3. Execute the query

```
class IntroToLINQ
{
    static void Main()
    {
        int[] numbers = new int[7] { 0, 1, 2, 3, 4, 5, 6 }; //  1. Data
source.

        // 2. Query creation.
        // numQuery is an IEnumerable<int>
        var numQuery =
            from num in numbers
            where (num % 2) == 0
            select num;

        foreach (int num in numQuery) // 3. Query execution.
        {
            Console.Write("{0,1} ", num);
        }
    }
}
```

Notice, how we can write LINQ which looks SQL like but it is not.

LINQ to SQL

- By using LINQ to SQL, you can use the LINQ technology **to access the SQL** database just as you would access an in-memory collection.
- So, it is a component of the .NET Framework version 3.5 that provides a **run-time infrastructure for managing relational data as objects.**
- In LINQ to SQL, the data model of a relational database is mapped to an object model expressed in the programming language of the developer.
- When it runs, LINQ to SQL translates into SQL the language-integrated queries in the object model and sends them to the database for execution. When the database returns the result, LINQ to SQL translates them back to objects.
- In short, when you write in LINQ, it translate to SQL. So, you do not even need to know how to write SQL.
- What we will be doing in the later labs is something called **LINQ to entities.**

Why LINQ beats SQL

Generally, when querying databases, LINQ is in most cases a **significantly more productive** querying language than SQL.

LINQ is tidier and higher level.

For example.

```
SELECT UPPER(Name) FROM
(
    SELECT *, RN = row_number()
    OVER (ORDER BY Name)
    FROM Customer
    WHERE Name LIKE 'A%'
) A
WHERE RN BETWEEN 21 AND 30
ORDER BY Name
```

Query in SQL

```
var query =
    from c in db.Customers
    where c.Name.StartsWith("A")
    orderby c.Name
    select c.Name.ToUpper();

var thirdPage =
query.Skip(20).Take(10);
```

Query in LINQ

Interesting C# operators

?? Operator

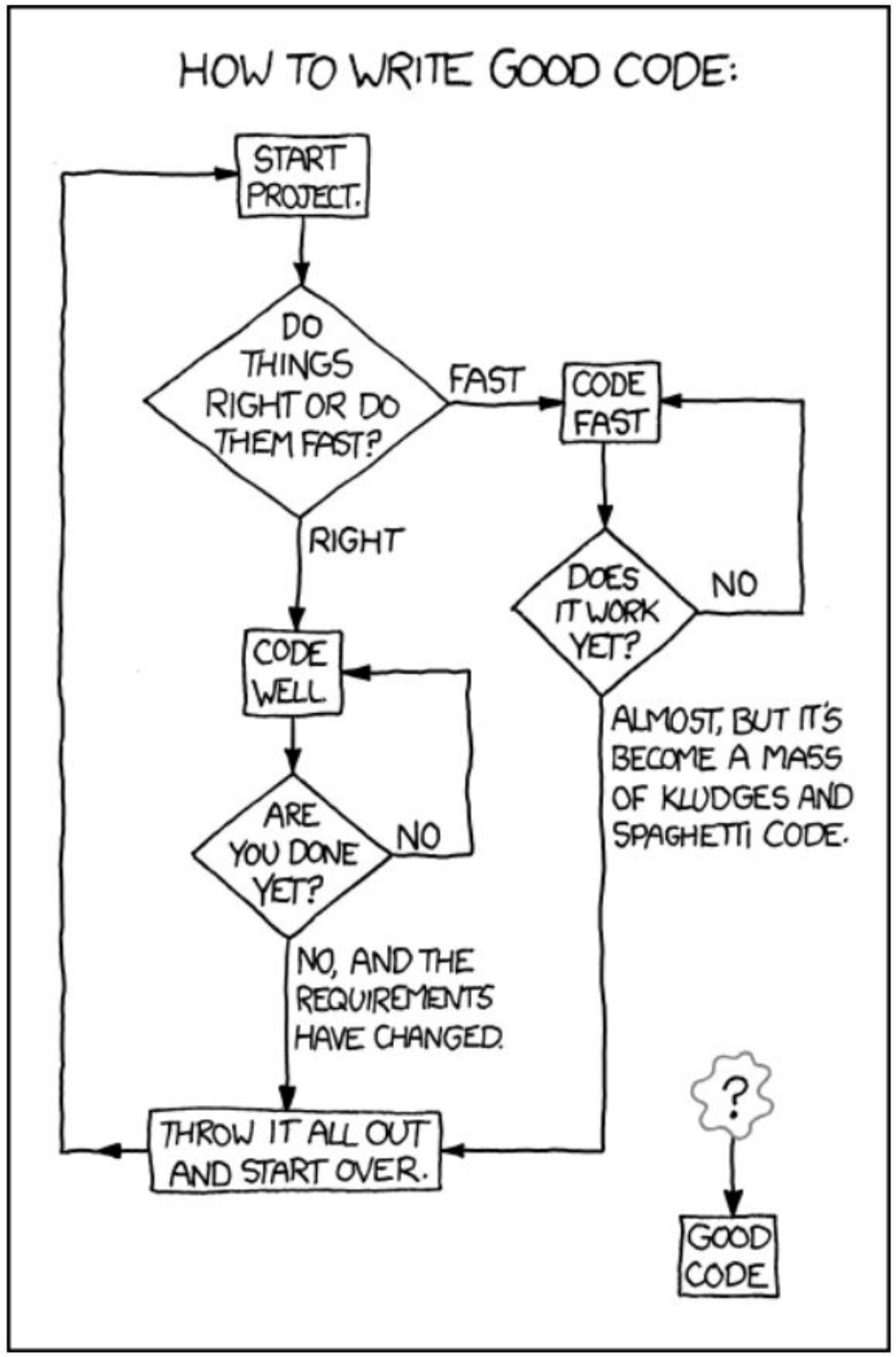
The ?? operator is called the null-coalescing operator. It returns the left-hand operand if the operand is not null; otherwise it returns the right hand operand.

?. (Null-conditional operator)

Tests the value of the left-hand operand for null before performing a member access (?.) or index (?[]) operation; returns null if the left-hand operand evaluates to null.

=> Operator

The => token is called the lambda operator. It is used in lambda expressions to separate the input variables on the left side from the lambda body on the right side.



For this subject, we strife for completion of a feature instead of the writing of good code. Even though writing good code is a good practice, often time it comes with experience.

In other words, the completion of a specific feature in the assignment takes precedence over how good the codes are……..

This is evident in real life as well where the completion of a feature is deemed to be more important. (But you should honestly try to improve over time)