**Type System**

* **Strongly-typed language, compiler check code is type safe when CLR allocates and reclaims memory, compiler error when integer + string.**
* **Value types: System.Object -> system.ValueType, memory is allocated for the value on stack of the method (call stack). Stack is cleared when method is returned.**
* **Reference types: string, class, delegate, array or interface, default is null, value (the value types it contains) is allocated on the managed heap, and the reference is allocated on the stack.** GC kicks off when reach memory threshold, it stops all and remove unused objects on heap and re-organise (adjust pointers) objects in use.
* Method is not in Stack or Heap. It’s in Code Segment within the assembly, and then gets copied to process memory Just-in-time complier.
* C# compilation: convert C# to Common intermediate Language (CIL, Assembly code, defined by Common Language Infrastructure (CLI));
* C# run: use just-in-time complier to convert CIL (complied code) to native machine code for operation system to run.
* Common Language Runtime: C# Complier, GC, Security, JIT Complier, Exception handler, Type Safety, Thread support.

**Cast**

* **Implicit Conversion: int (32-bit) to long (64-bit); base class to derived class.**
* **Explicit Conversion (Cast): double (64-bit float) to int; derived class to base class.**
* **Boxing/Unboxing: Convert a value type to reference type and vice versa, it happens in the compile time, so exception will throw. It’s expansive too. It is useful when using reflections, because it doesn’t know the type of the method returns until run-time.**
* **BitConverter to convert byte[], TeyParse to convert string to int, Convert Class to convert any type.**
* **Dynamic types and members will not be resolved until runtime.**
* **Implicit type: var types decided at compile time, var will no cause boxing because var uses the type when the variable is declared.**

**Class**

* **Struct for lightweight objects such as point, constructor needs to have parameters, no class inheritance for struct but it can implement interface.**
* **Inheritance: “Is-a” relationship. Derived class gain all the members from base class except constructor and finalizer.**
* **Abstract method has to be override, virtual method can be override, sealed can’t.**
* **Abstract class can’t be instantiation, it has to be inherited. Calls must be abstract if it has abstract member and all abstract methods have to be implemented in derived class.**
* **Interface with all abstract members and can have one-to-many relationship with class. It used to define specific capabilities.**
* **Polymorphism: Object type changes at the run time, override methods do difference things at the run time.**
* **Fields cannot be virtual; only methods, properties, events and indexers can be virtual.**
* **“New” hides the methods, so if the object type is base class, it will call base method. “Override” overrides the method; it allows object to access derived method.**
* **“public”: same assembly or referenced assemblies. “private”: same class/struct. “protected”: same or derived class (cross assembly). “interal”: same assembly. “protected internal”: same assembly and derived class (cross assembly). “private protected”: same or derived class (not cross assembly).**
* **Field: should be private or protected. Properties: come with get and set methods.**
* **Extension method: method with “this” parameter.**
* **Finalizers: also called destructors, perform final clean-up when GC kicks in; use “~” in front of the class name. Most derived class get called first.**

**Tuples**

* **Combine multiple values with difference type without creating a class, like a data row.**
* **Method with Item1, Item2, Rest.Item1.**

**Interfaces**

* Interfaces can contain events, indexers, methods, and properties.
* An interface is like an abstract base class.
* If multi interfaces are used, derived class can have interface.method() to specify the implementation.

**Indexer**

* **Public string this[ing i]{ get{return page[i]}set{page[i] = value}}, indexer can be overloaded.**

**Generics (GenericClass<T>)**

* **Ability to design classes and method without defining object types at the compile time. Advantage is to avoid duplicate logic written just because the object types are different. It gives reusability, type safe (avoid casting or boxing).**
* **Constraints are useful to limit the types. GenericList<T> where T: struct,class,new(),object.**
* **Generic Method: public add<T>(T left,T right){return left+right;}**

**Delegates (late bindings)**

* **Define a reference type for a method but no content, it can be seen as a placeholder for a/some method(s), typical use is of course events.**
* **Multicast delegates are delegates where multiple methods have been chained together.**
* **public delegate void Del(string message);**

**public static void DelegateMethod(string message);**

**Del handler = DelegateMethod;**

**public void MethodWithCallback(int param1, int param2, Del callback) //pass in as parameter**

* **Multi-cast delegate combines delegate two or more delegates**
* **Event: void OnEventRaised(object sender, EventArgs args);**