1. Assemblies are auto-descriptive Packages of Types (class, enums, etc.)
2. Namespaces (and ‘using’ keyword use to avoid using the full name of Types in code).
3. ValueType vs. ReferenceType
   1. Boxing (creating an object around an Int32) and Unboxing (extracting the ValueType from the Box) is done automatically.
4. Fields/Methods : static vs. Instance.
5. Unit tests (Arrange/Act/Assert)
6. Object paradigm :
   1. Inheritance
      1. Layout of instances in memory.
      2. Instance Methods actually are static methods that accepts an implicit ‘this’ parameter.
   2. GetType() that gives the running instance type.
      1. using ‘is’ and ‘as’ keyword to test the running type (this respects the Liskov Substitution Principle)
   3. virtual/override
   4. Virtual Method Tables (a call to a virtual method occurs one indirection based on the running type of the instance)
   5. ‘sealed’ keyword (to forbid override)
7. Interface are Contracts
   1. Abstract classes can offer base implementation (but recall that a class can only have one base class).
   2. Interface members can be explicitly implemented.
      1. Enables support of different returned types (for identical parameters)
      2. Enables to « close » an implementation (like ‘sealed’ keyword can do).
8. IDisposable
   1. Acquire & Release as soon as possible
9. ‘using’ keyword is a syntactic sugar that guaranties try {…} finally { Dispose ! }
10. IEnumerable & IEnumerator
    1. ‘foreach’ keyword is a syntactic sugar for GetEnumerator/MoveNext/Current/Dispose.
    2. IEnumerable<T> implementation
       1. Only one method : GetEnumerator()
       2. IEnumerator<T> offers MoveNext() and Current property.
11. Generics
    1. Looks like C++ templates (but are actually quite different)
    2. Easy to use (thanks to type inference & intellisense)
    3. Not so easy to implement except for the simple “collections” like List<T> or Dictionary<TKey,TValue>
    4. Useful helpers:
       1. default(T) 🡪 default value of the type T (0, false, DateTime.MinValue, null, etc.)
       2. EqualityComparer<T>.Default and Comparer<T>.Default are automagically created singletons that offers equality and comparison for values of type T.
12. Explicit interface implementation
    1. Enables resolution of name clashes
       1. This is often required to support non-generic (object based) base interfaces like (for instance), in IEnumerable<T>/IEnumerator<T> that support “old” IEnumerable/IEnumerator
    2. Hides implemented method from intellisense
13. Dictionary<T> implementation
    1. Object.GetHashCode()
    2. Object.Equals
14. Design Pattern « Strategy » : IEqualityComparer<T>
15. Linked List (« Single Linked List » for our ITIDictionary<TKey,TValue>)
16. Yield return
17. MiniLinq : Extension methods at work
18. Func<T> , Action<T>, etc.
19. Linq in action : Ordering & Grouping
20. Observer pattern
    1. With interfaces and manual registration/unregistration
    2. With delegates
       1. Delegate.Combine does the dirty work : a combined delegate IS\_A delegate…
    3. With standard event support
       1. Standard signature : object source, EventArgs e
       2. ‘event’ keyword protects the delegate (only +=/-= are exposed)
21. Streams
    1. An easy to specialize abstract base class
       1. “Rich interface” that combines Read, Write & Seek capabilities “protected” by (CanRead/CanWrite/CanSeek properties)
       2. Handles byte[] buffers
    2. The 2 kind of Streams
       1. “Final” streams (FileStream, NetworkStream, MemoryStream, etc.)
       2. Decorator streams that wrap/decorate one or more other streams (see our TeeStream, KrabouilleStream)
22. Strings & Unicode
    1. Internal string representation is UTF-16
    2. The Encoding class is the “key” to understand encoding (byte[] 🡨🡪 string)
    3. Unicode is complex (it handles composition of characters – see the CharsAndStrings fixture)