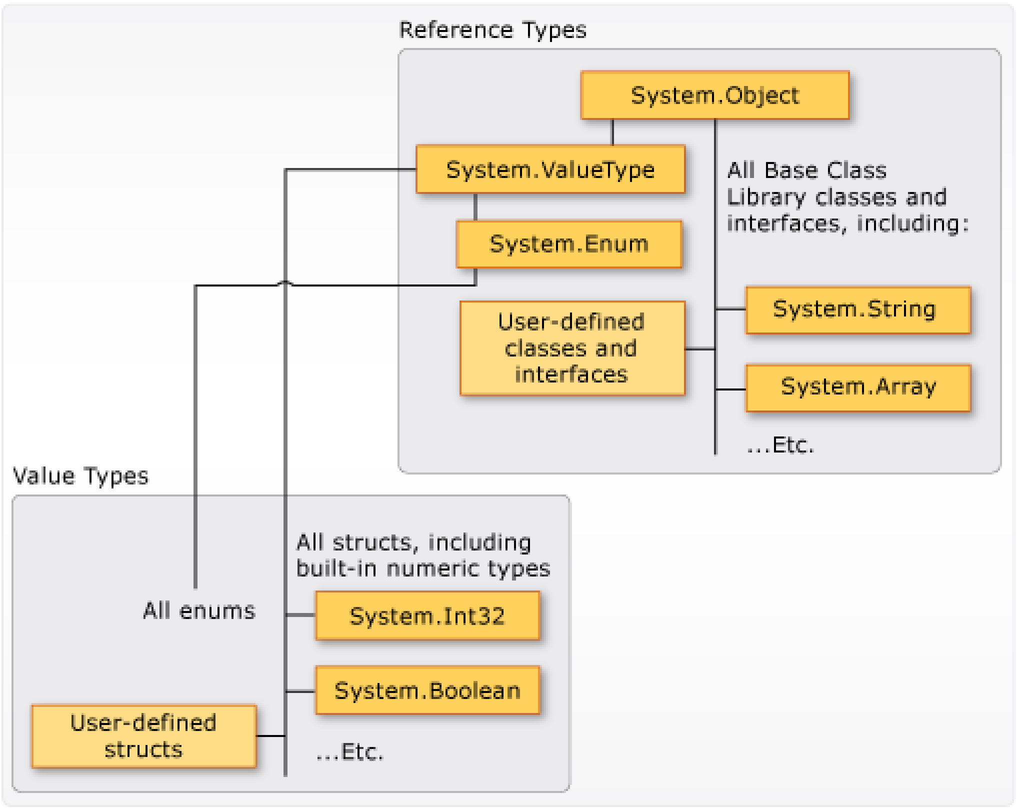
Essential Types and Data Structures in .NET – Questions and Answers

1. What is the difference between reference types and value types?

All types, including built-in numeric types such as System.Int32 (C# keyword: int), derive ultimately from a single base type, which is System.Object (C# keyword: object). This unified type hierarchy is called the Common Type System (CTS).

Each type in the CTS is defined as either a *value type* or a *reference type*. The following illustration shows the relationship between value types and reference types in the CTS.



Value types

Value types derive from System.ValueType, which derives from System.Object. Value type variables directly contain their values, which means that the memory is allocated inline in whatever context the variable is declared. Value type instances are usually allocated on a thread's stack (although they can also be embedded as a feld in a reference type object). There is no separate heap allocation or garbage collection overhead for value-type variables.

Value types are *sealed*, which means, for example, that you cannot derive a type from System.Int32, and you cannot define a struct to inherit from any user-defined class or struct because a struct can only inherit from System.ValueType. However, a struct can implement one or more interfaces. You can cast a struct type to an interface type; this causes a *boxing* operation to wrap the struct inside a reference type object on the managed heap.

Reference types

A type that is defined as a class, delegate, array, or interface is a *reference type*. When the object is created, the memory is allocated on the managed heap, and the variable holds only a reference to the location of the object. Types on the managed heap require overhead both when they are allocated and when they are reclaimed by the automatic memory management functionality of the CLR, which is known as *garbage collection*. However, garbage collection is also highly optimized, and in most scenarios it does not create a performance issue.

Source:
https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/types/

CLR via C#, Fourth Edition

2. What is the boxing and unboxing. Provide an example.

Boxing is the process of converting a value type to the type `object` or to any interface type implemented by this value type. When the CLR boxes a value type, it wraps the value inside a System.Object and stores it on the managed heap.

Unboxing extracts the value type from the object. An unboxing operation consists of:

- Checking the object instance to make sure that it is a boxed value of the given value type.
- Copying the value from the instance into the value-type variable.

Boxing is implicit; unboxing is explicit.

```
int i = 123;      // a value type
object o = i;     // boxing
int j = (int)o;   // unboxing
```

In relation to simple assignments, boxing and unboxing are **computationally expensive processes**. When a value type is boxed, a new object must be allocated and constructed. To a lesser degree, the cast required for unboxing is also expensive computationally. For more information, see Performance.

Source:
https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/types/boxing-and-unboxing

3. What is the immutability

Value types and mutability

A type is said to be immutable if it’s designed so that an instance can’t be changed after it’s been constructed. Immutable types often lead to a cleaner design than you’d get if you had to keep track of what might be changing shared values—particularly among different threads.

Immutability is particularly important for value types; they should almost always be immutable. Most value types in the framework are immutable, but there are some commonly used exceptions—in particular, the Point structures for both Windows Forms and Windows Presentation Foundation are mutable.

If you need a way of basing one value on another, follow the lead of DateTime and TimeSpan—provide methods and operators that return a new value rather than modifying an existing one. This avoids all kinds of subtle bugs, including situations where you may appear to be changing something, but you’re actually changing a copy. **Just say No to mutable value types.**

The String object is immutable. Every time you use one of the methods in the System.String class, you create a new string object in memory, which requires a new allocation of space for that new object. In situations where you need to perform repeated modifications to a string, the overhead associated with creating a new String object can be costly. The System.Text.StringBuilder class can be used when you want to modify a string without creating a new object. For example, using the StringBuilder class can boost performance when concatenating many strings together in a loop.

Source:
Jon Skeet – C# in Depth, 3rd Edition (Chapter 4.2.1 Intoducing Nullable<T>)

https://docs.microsoft.com/en-us/dotnet/standard/base-types/stringbuilder

4. Which data structures is applied when dealing with a recursive function?

Stack. Using LIFO, a call to a recursive function saves the return address so that it knows how to return to the calling function after the call terminates.

5. What are the differences between IEnumerable and IQueryable?

What IQueryable<T> has that IEnumerable<T> doesn't are two properties in particular—

one that points to a query provider (e.g., a LINQ to SQL provider) and another one pointing to a query expression representing the IQueryable<T> object as a runtime-traversable abstract syntax tree that can be understood by the given query provider

6. What's the difference between the System.Array.CopyTo() and System.Array.Clone()

CopyTo require to have a destination array when Clone return a new array.

CopyTo let you specify an index (if required) to the destination array.

7. What is the hash function?

A hash function is an algorithm that returns a numeric hash code based on a key.

A hash function must always return the same hash code for the same key.

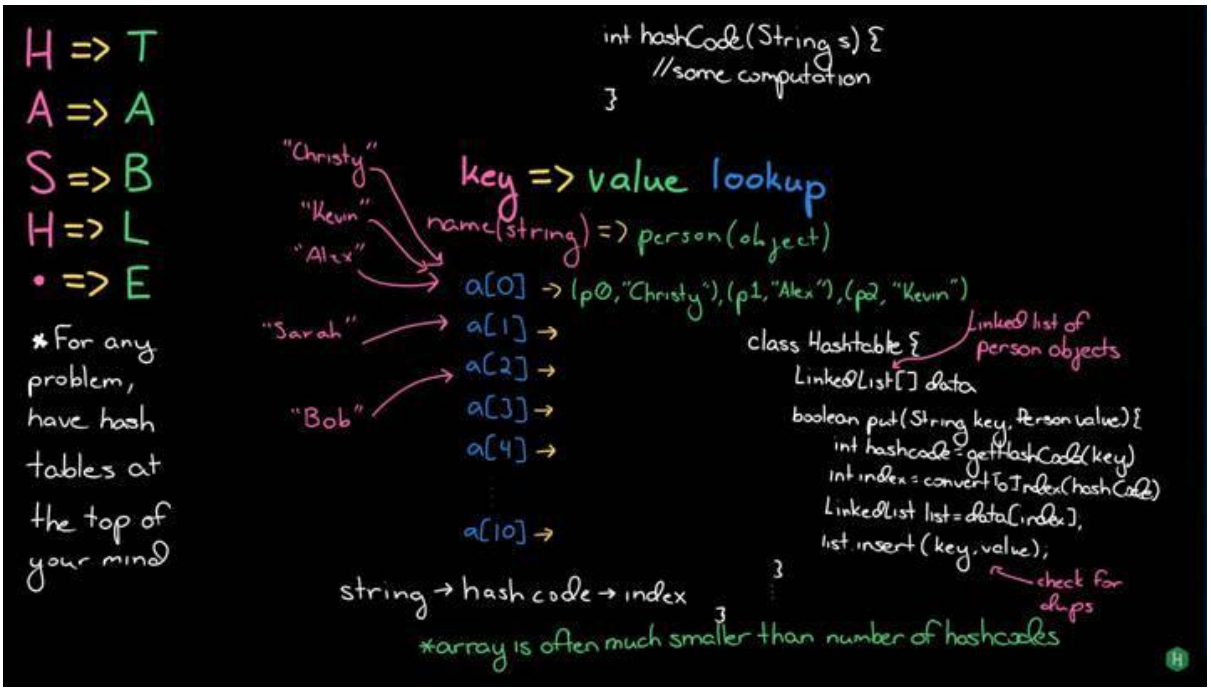
It is possible for a hash function to generate the same hash code for two different keys, but a hash function that generates a unique hash code for each unique key results in better performance when retrieving elements from the hash table.

One of the example of hash function is Jenkin's one_at_a_time hash:

```
uint32_t jenkins_one_at_a_time_hash(const uint8_t* key, size_t length) {
  size_t i = 0;
  uint32_t hash = 0;
  while (i != length) {
    hash += key[i++];
    hash += hash << 10;
    hash ^= hash >> 6;
  }
  hash += hash << 3;
  hash ^= hash >> 11;
  hash += hash << 15;
  return hash;
}
```

Source:
https://en.wikipedia.org/wiki/Hash_function

8. What is the Hashtable?



```
int hashCode(String s) {
    //some computation
}
```

```
class Hashtable {
    LinkedList[] data
    boolean put(String key, Person value) {
        int hashcode = getHashCode(key)
        int index = convertToIndex(hashCode)
        LinkedList list = data[index];
        list.insert(key, value);
    }
}
```

H => T
A => A
S => B
H => L
• => E
* For any problem, have hash tables at the top of your mind
key => value lookup
name(string) => person(object)
"Christy"
"Kevin"
"Alex"
"Sarah"
"Bob"
a[0] → (p0,"Christy"),(p1,"Alex"),(p2,"Kevin")
a[1] →
a[2] →
a[3] →
a[4] →
a[10] →
Linked list of person objects
check for dups
string → hash code → index
*array is often much smaller than number of hashcodes

Represents a collection of key/value pairs that are organized based on the hash code of the key.

Different keys could have the same hash code because there are infinite number of keys but finite number of hash codes – we have collisions

Then, each slot of a hash table is associated with (implicitly or explicitly) a set of records, rather than a single record. For this reason, each slot of a hash table is often called a *bucket*, and hash values are also called *bucket listing* or a *bucket index*.

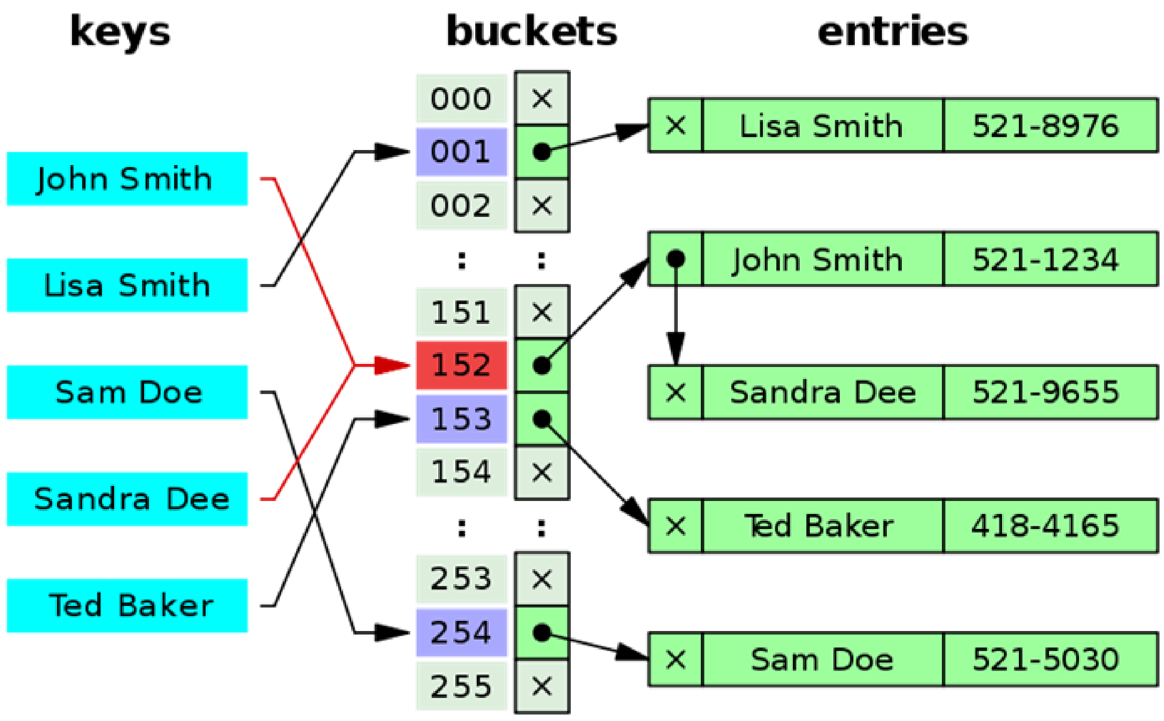
When an element is added to the Hashtable, the element is placed into a bucket based on the hash code of the key.

The load factor of a Hashtable determines the maximum ratio of elements to buckets. Smaller load factors cause faster average lookup times at the cost of increased memory consumption.

The default load factor of 1.0 generally provides the best balance between speed and size. A different load factor can also be specified when the Hashtable is created.

As elements are added to a Hashtable, the actual load factor of the Hashtable increases.

When the actual load factor reaches the specified load factor, the number of buckets in the Hashtable is automatically increased to the smallest prime number that is larger than twice the current number of Hashtable buckets.



Fast inserting, finding, and deleting - for a good hash table, this is O(1). For a terrible hashtable with a lot of collisions, this is O(n).

Source:
https://docs.microsoft.com/en-us/dotnet/standard/collections/hashtable-and-dictionary-collection-types

https://docs.microsoft.com/en-us/dotnet/api/system.collections.hashtable?view=netframework-4.7.1

https://www.youtube.com/watch?v=shs0KM3wKv8

https://en.wikipedia.org/wiki/Hash_table

9. What's the difference between Hashtable and Dictionary?

Hashtable is non-generic collection to store items as key/value pairs for quick look-up by key based on the hash code of the key. Dictionary<TKey,TValue> is the same but generic collection.

10. What is the HashSet?

Represents a set of values. The HashSet<T> class is based on the model of mathematical sets and provides high-performance set operations similar to accessing the keys of the Dictionary<TKey,TValue> or Hashtable collections.

In simple terms, the HashSet<T> class can be thought of as a Dictionary<TKey,TValue> collection without values.

A HashSet<T> collection is not sorted and cannot contain duplicate elements.

HashSet<T> provides many mathematical set operations, such as set addition (unions) and set subtraction. The following table lists the provided HashSet<T> operations and their mathematical equivalents.

HashSet(Of T) operation	Mathematical equivalent
UnionWith	Union or set addition
IntersectWith	Intersection
ExceptWith	Set subtraction
SymmetricExceptWith	Symmetric difference

Sources:
https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.hashset-1?view=netframework-4.7.1

https://stackoverflow.com/questions/4558754/define-what-is-a-hashset

https://msdn.microsoft.com/en-us/library/bb359438.aspx

11. When we should create structs?

If it logically represents a single value, consumes 16 bytes or less of storage, is immutable, and is infrequently boxed.

Source:
Essentials C# 6.0 (Chapter 8: Value Types)

12. Difference between Equality Operator (==) and Equals() Method in C#

Both the == Operator and the Equals() method are used to compare two value type data items or reference type data items. The Equality Operator (==) is the comparison operator and the Equals() method compares the contents of a string. The == Operator compares the reference identity while the Equals() method compares only contents.

13. Difference between const and readonly

Const can not be changed after initialization, readonly can be changed in constructor. Also readonly is shallow copy and elements of readonly array can be changed.

14. What's the difference between the 'ref' and 'out' keywords?

The ref modifier means that:

- The value is already set and
- The method can read and modify it.

The out modifier means that:

- The Value isn't set and can't be read by the method until it is set.
- The method must set it before returning.