

Essential Types & Data Structures in .NET

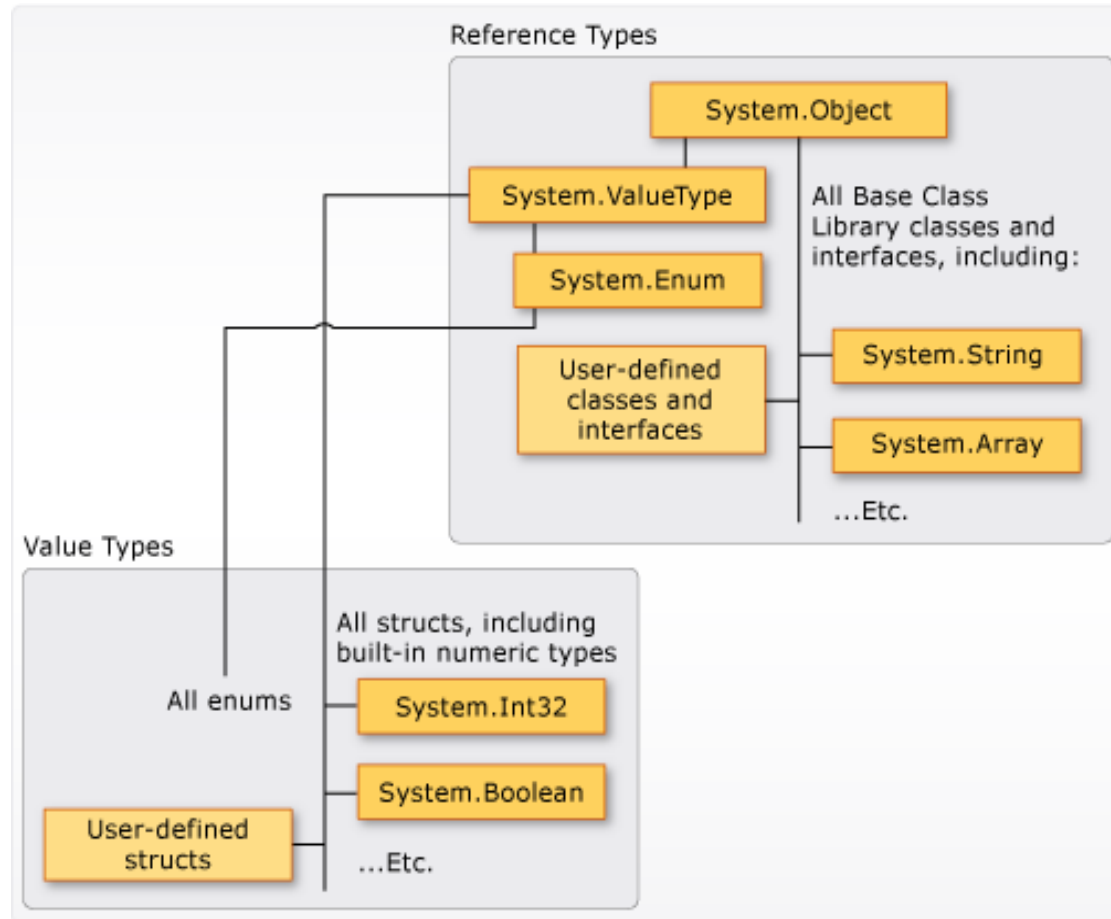
Value and Reference Types
Boxing and Unboxing
Immutability

Value Types and Reference 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 and Reference Types



Value Types

Value type variables directly contain their values

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 field 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](#)

You cannot define a struct to inherit from any user-defined class or struct because a struct can only inherit from [System.ValueType](#).

Struct can implement one or more interfaces. You can cast a struct type to an interface type.

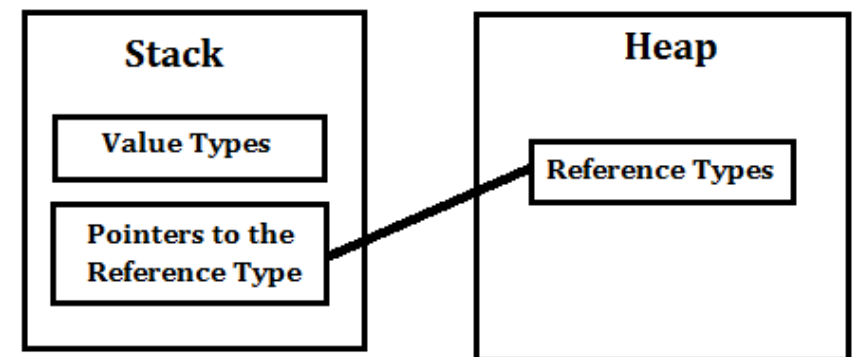
Reference Types

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.



Call Stack

You can think of a stack as a collection of boxes stacked on top of each other

- LIFO principle (Last In First Out)
- You can only use the top
- When the top is not needed – throw it away, the top becomes the box under it
- A new box is placed on top and becomes the top

Call stack

- Default size is fixed: 1MB (32bit), 4MB (64bit)
 - memory is allocated once when the thread is created
- Boxes are callable methods
 - in the box - variables
 - arguments, local variables, return value
 - The formal name of the box is a stack frame
 - is very fast!
 - adding / removing - just moving the pointer to the top of the stack
 - is static. Sizes and data types are determined during compilation



Managed Heap

You can think of a stack as a collection of boxes placed on the floor

- You can use any box
- If the box is no longer needed
 - janitor will take it away (garbage collector)
- Each box has its own address in the heap
- stored in virtual memory
- Boxes are separate objects
- **Large size** ~1.5GB (32bit), ~8TB (64bit)
 - requested from the OS as needed
- Is unique per process, not per thread (problems with concurrent access)
- Is **slower**
 - memory is allocated every time you add new object
 - cleaned up by the garbage collector (GC), whose work takes time
- Is **dynamic**. The size of the data is determined when executing



Boxing and Unboxing

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.



Boxing and Unboxing

// a value type

```
int stackVar = 123;
```

// boxing

```
object boxedVar = stackVar;
```

// unboxing

```
int unBoxed = (int)boxedVar;
```

On the Stack

stackVar

12

```
int stackVar = 12
```

boxedVar

```
object boxedVar = stackVar
```

unBoxed

12

```
int unBoxed = (int)boxedVar
```

On the Heap

(stackVar boxed)

int

12

@dotnet-tricks.com

Boxing and unboxing

Boxing and 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](#).

Immutable Types

A type is said to be immutable if it's designed so that an instance can't be changed after it's been constructed.

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.

Immutable Types

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.

You should create structs if it logically represents a single value, consumes 16 bytes or less of storage, is immutable, and is infrequently boxed.

Just say NO to mutable value types



String

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.

Other immutable types

Tuple Class

- data structure that has a specific number and sequence of elements
- is a reference type
- != tuple type ([System.ValueTuple](#))

```
var population = Tuple.Create("New York", 7891957, 7781984, 7894862,  
7071639, 7322564, 8008278);
```

Init:

```
public class Friend  
{  
    public string FirstName { get; init; }  
    public string MiddleName { get; init; }  
    public string LastName { get; init; }  
}
```


Questions

What is the difference between reference types and value types?

What is the difference between call stack and managed heap?

What is the boxing and unboxing?

What is the immutability?

Arrays

Array, List, Queue, Linked List

IEnumerable and

IQueryable

Arrays

Definition:

You can store multiple variables of the same type in an array data structure. You declare an array by specifying the type of its elements.

Base points:

Array elements can be of any type, including an array type

An array can be [Single-Dimensional](#), [Multidimensional](#) or [Jagged](#).

Arrays are zero indexed: an array with n elements is indexed from 0 to n-1

Arrays

Single-Dimensional

- `int[] array = new int[5];`
- `int[] array1 = new int[] { 1, 3, 5, 7, 9 };`

Multidimensional

- `int[,] array = new int[4, 2];`
- `int[, ,] array1 = new int[4, 2, 3];`

Jagged

- `int[][] jaggedArray = new int[3][];`
- `jaggedArray[0] = new int[5];`
- `jaggedArray[1] = new int[4];`
- `jaggedArray[2] = new int[2];`

Arrays key points

The number of dimensions and the length of each dimension are established when the array instance is created.

Size can't be changed during the lifetime of the instance.

The default values of numeric array elements are set to **zero**, and reference elements are set to **null**

A jagged array is an array of arrays, and therefore its elements are reference types and are initialized to **null**

- Array types are [reference types](#) derived from the abstract base type [Array](#).

Implements [IEnumerable](#) and [IEnumerable<T>](#), so you can use [foreach](#) iteration

Class Array

Definition:

Provides methods for creating, manipulating, searching, and sorting arrays, thereby serving as the base class for all arrays in the common language runtime

Base points:

The Array class is **not part** of the [System.Collections](#) namespaces

Considered a collection because it is based on the [IList](#) interface.

By default, the **maximum** size of an Array is **2 gigabytes** (GB).

Class Array key points

In a 64-bit environment, you can avoid the size restriction by setting the **enabled** attribute of the [gcAllowVeryLargeObjects](#) configuration element to **true** in the run-time environment.

Array will still be limited to a total of **4 billion elements**, and to a maximum index of 0X7FEFFFFF in any given dimension

Some methods, such as [CreateInstance](#), [Copy](#), [CopyTo](#), [GetValue](#), and [SetValue](#), provide **overloads** that accept 64-bit integers as parameters to accommodate large capacity arrays

[LongLength](#) and [GetLongLength](#) return 64-bit integers indicating the length of the array

Class array. Async

Enumerating through a collection is intrinsically **not a thread safe** procedure.

Even when a collection is synchronized, other threads can still modify the collection, which causes the enumerator to throw an exception.

To guarantee thread safety during enumeration, you can either **lock** the collection during the entire enumeration or catch the exceptions resulting from changes made by other threads.

Class Array. Clone and CopyTo

Clone

- Method creates a **shallow** copy of an array. A shallow copy of an Array copies only the elements of the Array, whether they are reference types or value types, but it **does not** copy the objects that the references refer to. The references in the new Array point to the same objects that the references in the original Array point to.

CopyTo

- The Copy static method of the Array class copies a section of an **array** to **another array**. The CopyTo method copies all the elements of an array to another **one-dimension** array.

Both perform a shallow copy

If value type is used for CopyTo/Clone any change in the values made on them will not get reflected on the original whereas on reference type the change gets reflected.

List

Definition:

Represents a strongly typed list of objects that can be accessed by index. Provides methods to search, sort, and manipulate lists.

Base points:

Size is dynamically increased as required.

You can increase the maximum capacity to 2 billion elements on a 64-bit system by setting the **enabled** attribute of the configuration element to **true** in the run-time environment

List key points

In deciding whether to use the `List<T>` or [ArrayList](#) class, both of which have similar functionality, remember that the `List<T>` class performs better in most cases and is type safe.

If a value type is used for type T , you need to consider implementation and boxing issues in `ArrayList`

If a value type is used for type T , the compiler generates an implementation of the `List<T>` class specifically for that value type

Queue vs Stack

Definition:

Queue - Represents a first-in, first-out (FIFO) collection of objects.

Stack - Represents a variable size last-in-first-out (LIFO) collection of instances of the same specified type.

Basic points:

Queues and stacks are useful when you need temporary storage for information (that is, when you might want to **discard** an element after retrieving its value)

Use Queue if you need to access the information in the same order that it is stored in the collection.

Use [Stack](#) if you need to access the information in reverse order.

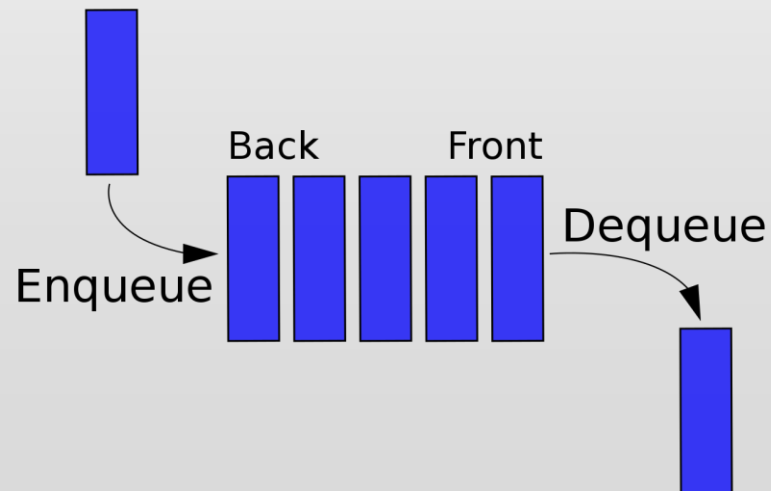
Queue key points

Three main operations can be performed on a Queue and its elements:

[Enqueue](#) adds an element to the end of the Queue.

[Dequeue](#) removes the oldest element from the start of the Queue.

[Peek](#) returns the oldest element that is at the start of the Queue but does not remove it from the Queue.



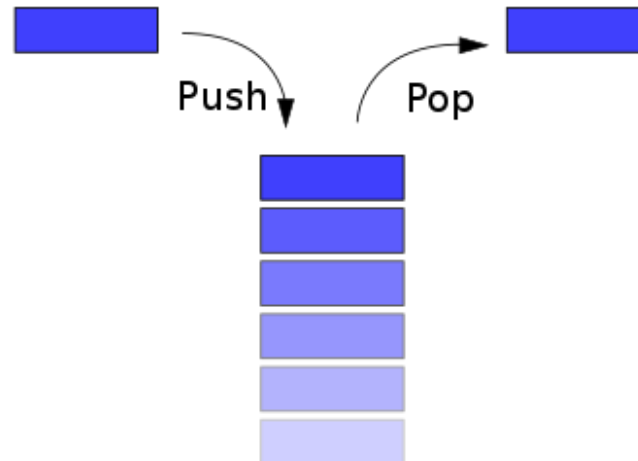
Stack key points

Three main operations can be performed on a `System.Collections.Generic.Stack<T>` and its elements:

[Push](#) inserts an element at the top of the [Stack](#).

[Pop](#) removes an element from the top of the `Stack<T>`.

[Peek](#) returns an element that is at the top of the `Stack<T>` but does not remove it from the `Stack<T>`.



Stack and Queue async

A `Queue<T>` and `Stack<T>` can support multiple readers concurrently, as long as the collection is not modified.

Enumerating through a collection is intrinsically not a thread-safe procedure.

To guarantee thread safety during enumeration, you can lock the collection during the entire enumeration.

To allow the collection to be accessed by multiple threads for reading and writing, you must implement your own synchronization.

LinkedList

Definition:

Represents a doubly linked list.

Basic Points:

It supports enumerators and implements the [ICollection](#) interface, consistent with other collection classes in the .NET Framework.

You can remove nodes and reinsert them, either in the same list or in another list, which results in no additional objects allocated on the heap

If the `LinkedList<T>` is empty, the [First](#) and [Last](#) properties contain **null**.

Linked list key points

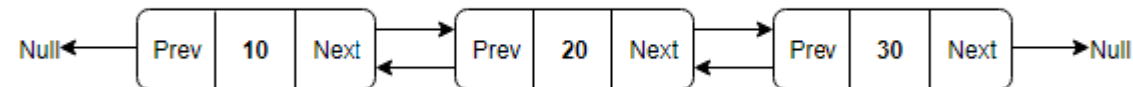
LinkedList<T> provides separate nodes of type [LinkedListNode<T>](#), so insertion and removal are **O(1)** operations.

List also maintains an internal count, getting the [Count](#) property is an **O(1)** operation.

Lists that contain reference types perform better when a node and its value are created at the same time

The LinkedList<T> class does not support chaining, splitting, cycles, or other features that can leave the list in an inconsistent state.

The list remains consistent on a single thread



LinkedList async

The only multithreaded scenario supported by `LinkedList<T>` is multithreaded read operations

If the `LinkedList<T>` needs to be accessed by multiple threads, you will need to implement their own synchronization mechanism.

Stack vs Queue vs List vs LinkedList

Pushing to Stack...	Time used:	7067 ticks
Poping from Stack...	Time used:	2508 ticks
Enqueue to Queue...	Time used:	7509 ticks
Dequeue from Queue...	Time used:	2973 ticks
Insert to List at the front...	Time used:	5211897 ticks
RemoveAt from List at the front...	Time used:	5198380 ticks
Add to List at the end...	Time used:	5691 ticks
RemoveAt from List at the end...	Time used:	3484 ticks
AddFirst to LinkedList...	Time used:	14057 ticks
RemoveFirst from LinkedList...	Time used:	5132 ticks
AddLast to LinkedList...	Time used:	9294 ticks
RemoveLast from LinkedList...	Time used:	4414 ticks

IEnumerable

Definition:

Exposes an enumerator, which supports a simple iteration over a non-generic collection

Basic points:

IEnumerable is the base interface for all non-generic collections that can be enumerated.

IEnumerable contains a single method, [GetEnumerator](#), which returns an [IEnumerator](#).

[IEnumerator](#) provides the ability to iterate through the collection by exposing a [Current](#) property and [MoveNext](#) and [Reset](#) methods.

It is a best practice to implement IEnumerable and IEnumerator on your collection classes to enable the foreach syntax, however implementing IEnumerable is not required.

IQueryable

Definition;

Provides functionality to evaluate queries against a specific data source wherein the type of the data is not specified.

Basic points:

IQueryable<T> extends the IEnumerable<T> interface, so anything you can do with a "plain" IEnumerable<T>, you can also do with an IQueryable<T>.

Queries that do not return enumerable results are executed when the [Execute](#) method is called

IEnumerable vs IQueryable

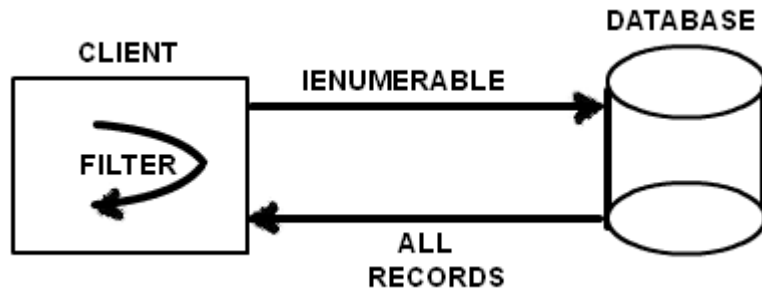
What IQueryable<T> **has** that IEnumerable<T> **doesn't** are two properties in particular:

That points to a query provider (e.g., a LINQ to SQL provider)

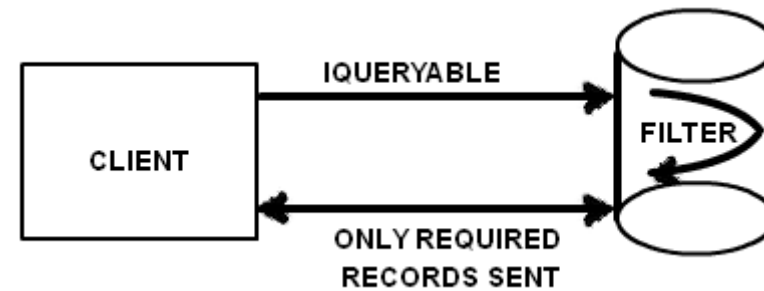
Pointing to a query expression representing the IQueryable<T> object as a runtime-traversable abstract syntax tree that can be understood by the given query provider

IEnumerable vs IQueryable

```
EmpEntities ent = new EmpEntities();  
IEnumerable<Employee> emp = ent.Employees;  
IEnumerable<Employee> temp = emp.Where(x => x.Empid == 2).ToList<Employee>();
```



```
EmpEntities ent = new EmpEntities();  
IQueryable<Employee> emp = ent.Employees;  
IQueryable<Employee> temp = emp.Where(x => x.Empid == 2).ToList<Employee>();
```



Questions

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

What are the differences between IEnumerable and IQueryable?

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

Hashtable and Dictionary Collection Types

Are you sure that you know HashTable?



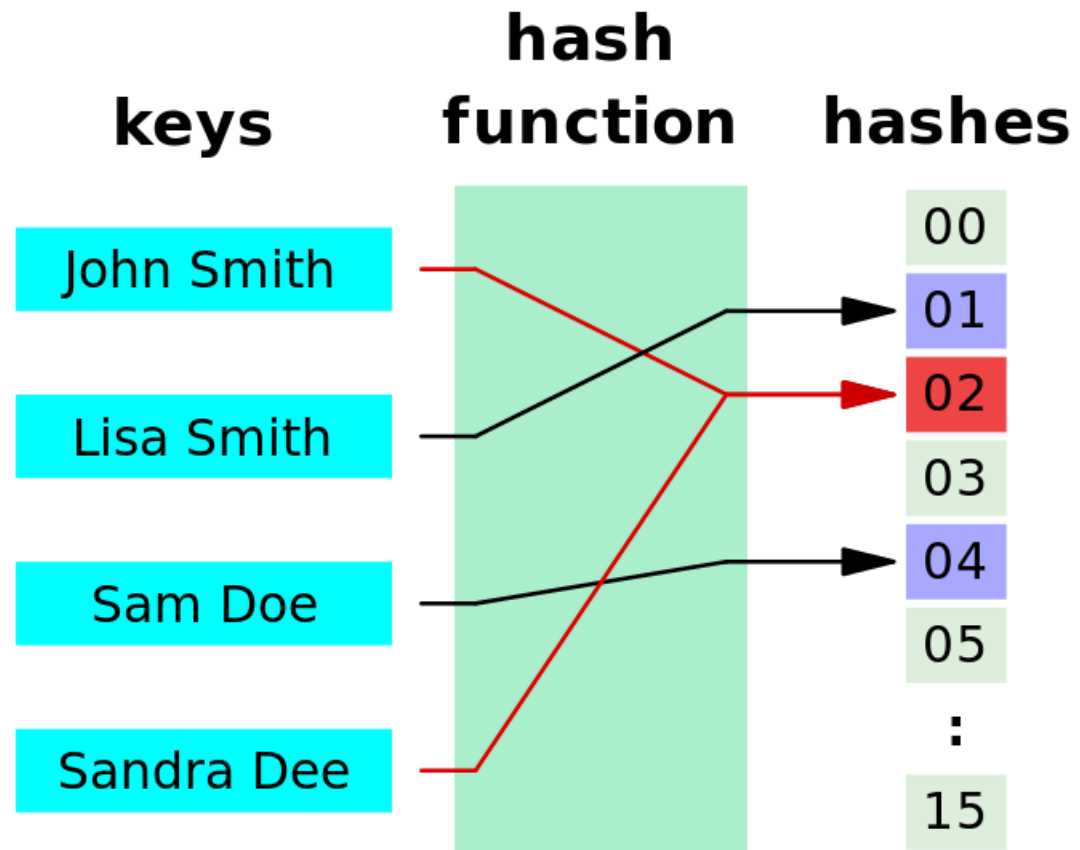
Hash Function

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.

Hash Function



Jenkins hash functions

- The collection of (non-[cryptographic](#)) [hash functions](#) for multi-[byte](#) keys designed by [Bob Jenkins](#)
- Jenkins'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;  
}
```

H => T

A => A

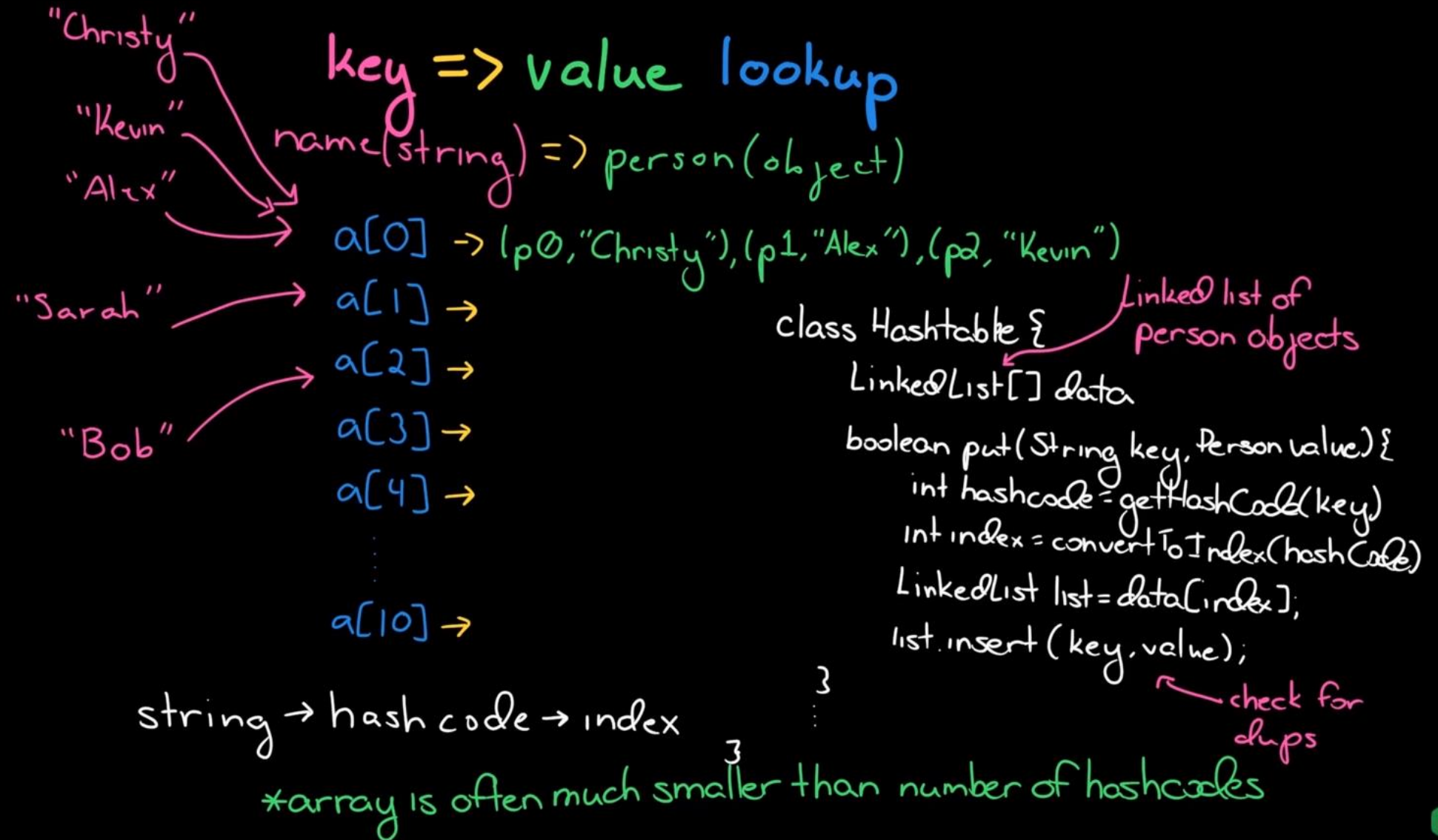
S => B

H => L

• => E

* For any problem, have hash tables at the top of your mind

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

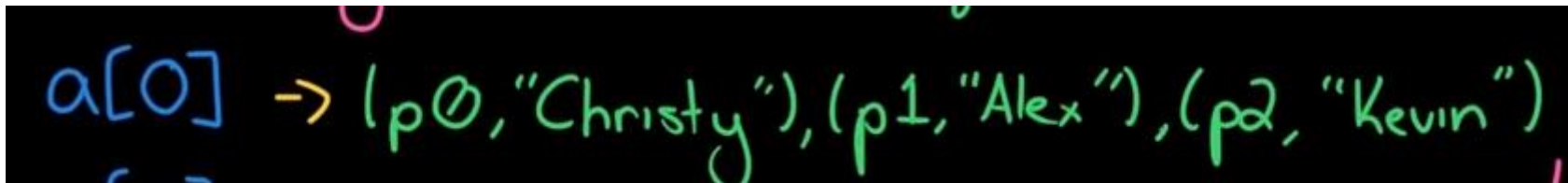


Hashtable

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*.



The image shows a handwritten code snippet on a black background. It depicts a bucket in a hash table, labeled 'a[0]', which points to a set of three records: (p0, "Christy"), (p1, "Alex"), and (p2, "Kevin"). The text is written in a light green color with a yellow arrow pointing from the bucket index to the list of records.

```
a[0] -> (p0, "Christy"), (p1, "Alex"), (p2, "Kevin")
```

Hashtable

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.

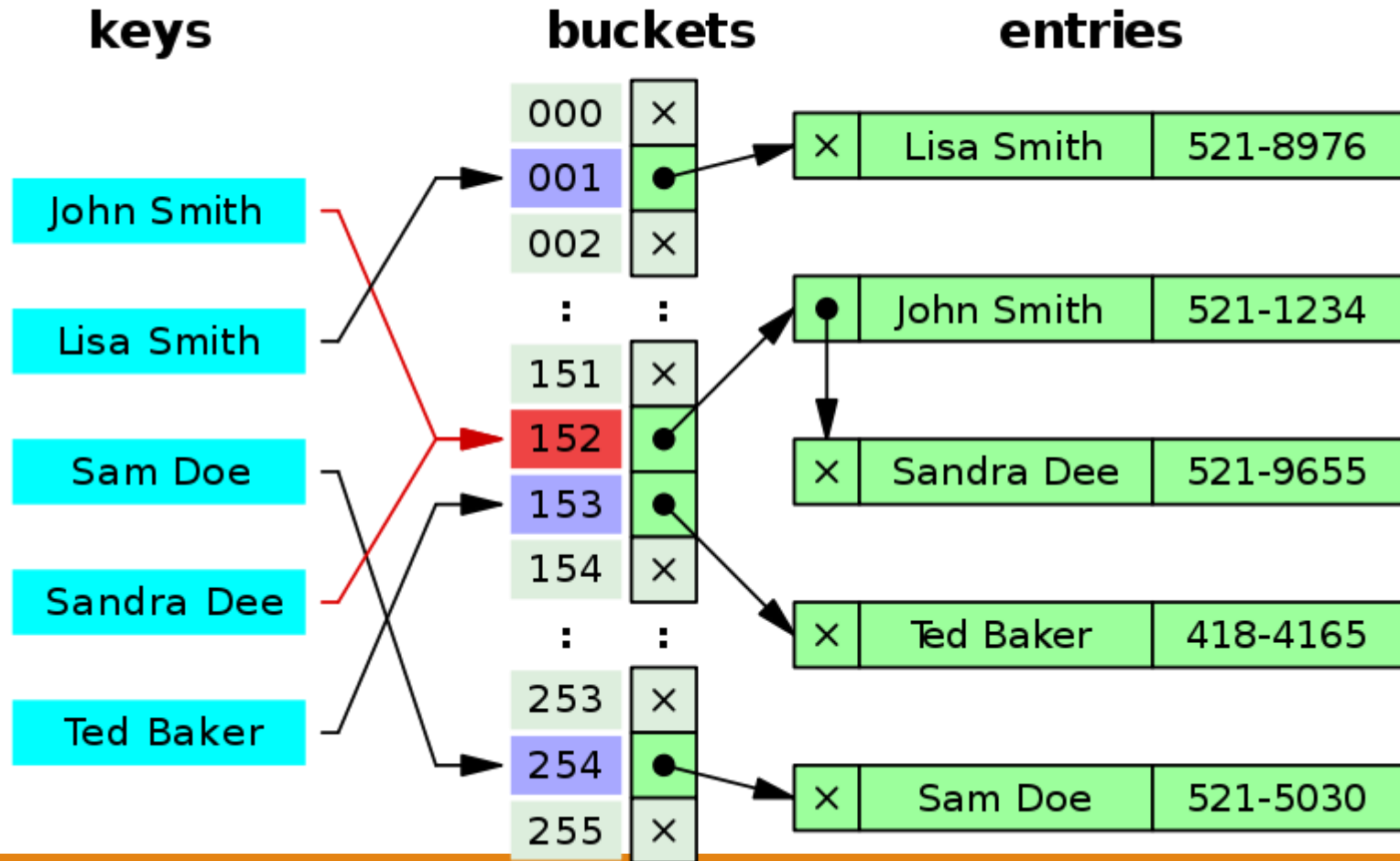
The default load factor of 1.0 generally provides the best balance between speed and size. A different load factor can also be specified when the [Hashtable](#) is created.

Hashtable

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.

Resolving collision - chaining with linked lists



Hashtable

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 $O(n)$.

Each element is a key/value pair stored in a [DictionaryEntry](#) object. A key cannot be **null**, but a value can be.

Key objects must be immutable as long as they are used as keys in the [Hashtable](#).

The objects used as keys by a Hashtable are required to override the [Object.GetHashCode](#) method (or the [IHashCodeProvider](#) interface) and the [Object.Equals](#) method (or the [IComparer](#) interface)

Override GetHashCode

If GetHashCode is not overridden, hash codes for reference types are computed by calling the Object.GetHashCode method of the base class, which computes a hash code based on an object's reference; for more information, see [RuntimeHelpers.GetHashCode](#).

.NET Framework does not guarantee the default implementation of the [GetHashCode](#) method, and the value this method returns may differ between .NET Framework versions and platforms, such as 32-bit and 64-bit platforms. For these reasons, do not use the default implementation of this method as a unique object identifier for hashing purposes.

Dictionary<TKey,TValue>

The [Dictionary<TKey,TValue>](#) generic class provides a mapping from a set of keys to a set of values.

Each addition to the dictionary consists of a value and its associated key.

Retrieving a value by using its key is very fast, close to $O(1)$, because the [Dictionary<TKey,TValue>](#) class is implemented as a hash table.

Dictionary<TKey,TValue>

[Dictionary<TKey,TValue>](#) requires an equality implementation to determine whether keys are equal.

You can specify an implementation of the [IEqualityComparer<T>](#) generic interface by using a constructor that accepts a **comparer** parameter

if you do not specify an implementation, the default generic equality comparer [EqualityComparer<T>.Default](#) is used.

If type **TKey** implements the [System.IEquatable<T>](#) generic interface, the default equality comparer uses that implementation.

HashSet<T>

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.

If order or element duplication is more important than performance for your application, consider using the [List<T>](#) class together with the [Sort](#) method.

HashSet<T>

```
HashSet<int> evenNumbers = new HashSet<int>();  
HashSet<int> oddNumbers = new HashSet<int>();  
  
for (int i = 0; i < 5; i++)  
{  
    // Populate numbers with just even numbers.  
    evenNumbers.Add(i * 2);  
  
    // Populate oddNumbers with just odd numbers.  
    oddNumbers.Add((i * 2) + 1);  
}
```


HashSet<T>

[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

Choosing between Hashtable, HashSet & Dictionary

I want to...	Generic collection options	Non-generic collection options	Thread-safe or immutable collection options
Store items as key/value pairs for quick look-up by key	Dictionary<TKey,TValue>	Hashtable (A collection of key/value pairs that are organize based on the hash code of the key.)	ConcurrentDictionary<TKey,TValue> ReadOnlyDictionary<TKey,TValue> ImmutableDictionary<TKey,TValue>
A set for mathematical functions	HashSet<T> SortedSet<T>	No recommendation	ImmutableHashSet<T> ImmutableSortedSet<T>

Questions

What is the hash function?

What's the difference between Hashtable and Dictionary?

What is the HashSet<T>?

The End

Thank you for attention!