# Goal 1: C# and .NET

## Able to use reference and value types correctly

<http://www.albahari.com/valuevsreftypes.aspx>

<https://www.tutlane.com/tutorial/csharp/csharp-value-type-and-reference-type-with-examples>

The **stack** is a first-in last-out memory structure with static memory allocation (happening at compile time). Access is very fast.

* The **reference** portion of reference-typed local variables and parameters (such as the **myTextBox** reference)
* **Value-typed** local variables and **method parameters** (structs, as well as integers, bools, chars, DateTimes, etc.)

The following data is stored on the **heap** (<https://en.wikipedia.org/wiki/Heap_(data_structure)>) with dynamic memory allocation (means happening runtime). Access is slower.

* The **content** of reference-type objects.
* Anything structured inside a reference-type object.

## Understand .NET collections

Reference: Developer Guide to Collections in .NET (book), <https://resources.oreilly.com/examples/9780735659278-files/tree/master/Developer's%20Guide%20to%20Collections> (Git repo)

### Basic collections

* **Array**

Proes: Quick access, do not need to implement (language feature)

Cons:

* **Linekd list**
  + **Singly linked**
  + **Doubly linked**
* **Associative arrays**
  + **Associative list**
  + **Hash table**
* …

### Collection interfaces:

**IEnumerable** – gives the ability to enumerate the collection, this make it possible to use the collection in **foreach** statement

**ICollection** – gives extra information about the collection, for example the **count**

**IList** – gives thepossibility to **index** the elements in the collection

# Javascript Basics

## Basic setup of JS runtime

**Heap** – memory allocation

**Stack** – execution contexts

**Callback Queue** – queue for function calls

**Event Loop** – manager for getting function calls out of the **callback queue** and pushing into the **stack**

**Web API** – like 3rd party staffs given by the browser (AJAX, setTimeout, click, etc…)

**Scope** - context environment (also known as lexical environment) created when a function is written. This context defines what other data it has access to

**Closure –**

1. Closures are functions that have access to variables from another function's scope. This is accomplished by creating a function inside another function.

2. A Closure is a function that returns another function.

3. A Closure is an implicit, permanent link between a function and its scope chain.

Examples:

Var Toaster = (function(){…})()

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

function firstName(first){  
 function fullName(last){  
 console.log(first + " " + last);  
 }

return fullName;  
}

var name = firstName("Mister");  
name("Smith") // Mister Smith  
name("Jones"); //Mister Jones

## Invoking a function

1. When function is invoked, it forms a new execution context

Execution context (?= scope chain) has a list of Variable- Activation Object list. Scope chain = Variable- Activation Objects + [[scope]]

[{Variable object},

{Activation object,

…}]

1. **Activation Object** is created which holds the declared variables, functions and parameters
2. Scope chain of **Activation Objects** are created. This is way to link or provide a systematic access to all variables and other functions that the current execution context (function in this case) has access to. **[[Scope]]** is the hidden mechanism that links these variable objects for identifier lookup. This hidden **[[Scope]]** is a property of the function, created at declaration, not invocation.

### Example

var firstNum = 1;

function number() {  
 var secondNum = 2;  
 return firstNum + secondNum;  
}

number();

In this case, number’s scope chain is linked to the global window object (the containing context that holds function number). This is what allows the engine to look outside of function number to find firstNum and secondNum.

## Related staffs

### Zone.js

A zone is an execution context that persists across async tasks, and allows the creator of the zone to observe and control execution of the code within the zone.

Angular uses the zone to patch async APIs(addEventListener, setTimeout(), ...) and uses notifications from these patched APIs to run change detection every time some async event happened.

**Stack Frame:** It is important to understand that a given stack frame can only be associated with one zone. (i.e. it is not possible for first half of a function to run in a different zone than later half of the function. It is possible that the same function will have different zone on different invocations). Zone can only be entered or left by entering or exiting Zone.prototype.run(). Zones updates stack traces to show zones for better visibility. Below are two stack snapshots from the above example, which show the associated zone for each stack frame.

**Key point:** When async work gets scheduled, the callback function will execute in the same zone as the zone which existed at the time of invoking the async API. This allows the zone to be tracked across many async invocations.

Zone have hooks to attach code to, and can be used to create profiling zones for example. (or LongStackTrace zone)

**Some usage**

* Associate some data with the zone, analogous to thread-local storage in other languages, which is accessible to any async operation inside the zone.
* Automatically track outstanding async operations within a given zone to perform cleanup or rendering or test assertion steps
* Time the total time spent in a zone, for analytics or in-the-field profiling
* Handle all uncaught exceptions or unhandled promise rejections within a zone, instead of letting them propagate to the top level