# C# Basics

## Goal 1: 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.

# 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.