Why and when use STATIC class:

When deciding whether to make a class static or non-static you need to look at what information you are trying to represent. This entails a more '**bottom-up**' style of programming where you focus on the data you are representing first. Is the class you are writing a *real-world object* like a rock, or a chair? These things are physical and have physical attributes such as color, weight which tells you that you may want to instantiate multiple objects with different properties. I may want a black chair AND a red chair at the same time. If you ever need two configurations at the same time then you instantly know you will want to instantiate it as an object so each object can be unique and exist at the same time.

On the other end*, static functions tend to lend more to actions* which do not belong to a real-world object or an object that you can easily represent. Remember that C#'s predecessors are C++ and C where you can just define global functions that do not exist in a class. This lends more to '**top-down**' programming. Static methods can be used for these cases where it doesn't make sense that an 'object' performs the task. By forcing you to use classes this just makes it easier to group related functionality which helps you create more maintainable code.

Most classes can be represented by either static or non-static, but when you are in doubt just go back to your OOP roots and try to think about what you are representing. Is this an object that is performing an action (a car that can speed up, slow down, turn) or something more abstract (like displaying output)?

*I would rather use static method if I want to use method, and use non-static method when I wish to create instance*

STATIC variables:

1. A static variable inside a function keeps its value between invocations.

(1) is the more foreign topic if you're a newbie, so here's an example:

#include <stdio.h>

void foo()

{

int a = 10;

static int sa = 10;

a += 5;

sa += 5;

printf("a = %d, sa = %d\n", a, sa);

}

int main()

{

int i;

for (i = 0; i < 10; ++i)

foo();

}

This print:

a = 15, sa = 15

a = 15, sa = 20

a = 15, sa = 25

a = 15, sa = 30

a = 15, sa = 35

a = 15, sa = 40

a = 15, sa = 45

a = 15, sa = 50

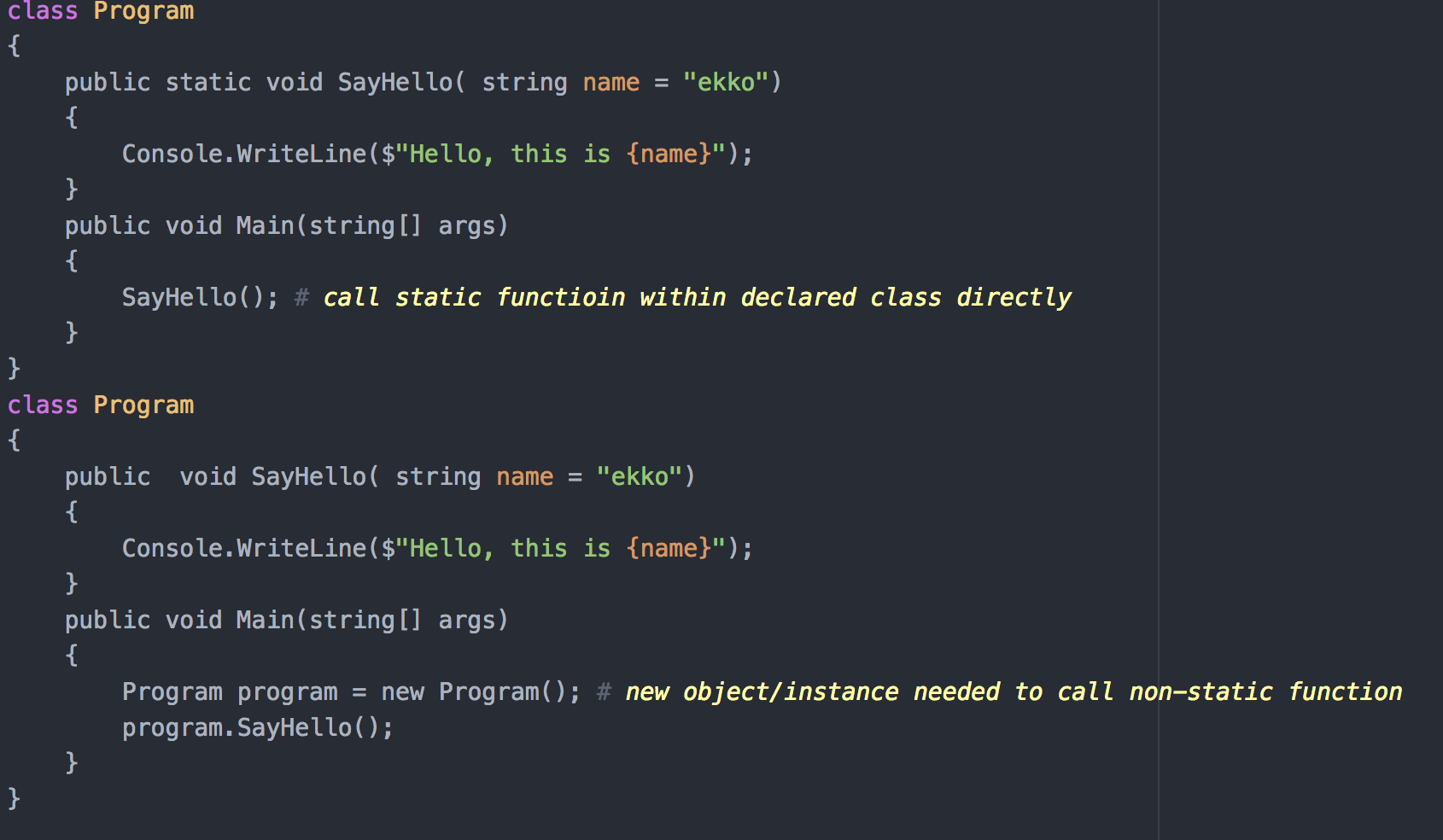
a = 15, sa = 55

a = 15, sa = 60

This is useful for cases where a function needs to keep some state between invocations, and you don't want to use global variables. Beware, however, this feature should be used very sparingly - it makes your code not thread-safe and harder to understand.

2.A static global variable or a function is *"seen"* only in the file it's declared in

In order to call non-static function, an object needs to be instantiated



*In the C programming language, static is used with global variables and functions to set their scope to the containing file. In local variables, static is used to store the variable in the statically allocated memory instead of the automatically allocated memory.*

**MEMORY ALLOCATION**

There are three types of allocation — static, automatic, and dynamic.

**Static Allocation** means, that the memory for your variables is allocated when the program starts. The size is fixed when the program is created. It applies to *global* variables, file scope variables, and variables qualified with static defined inside functions.

**Automatic memory allocation** occurs for (non-static) variables defined inside functions, and is usually stored on the stack (though the C standard doesn't mandate that a stack is used). You do not have to reserve extra memory using them, but on the other hand, have also *limited control over the lifetime* of this memory. E.g: *automatic variables in a function are only there until the function finishes.*

void func() {

int i; /\* `i` only exists during `func` \*/

}

**Dynamic memory allocation** is a bit different. You now control the exact size and the lifetime of these memory locations. If you don't free it, you'll run into memory leaks, which may cause your application to crash, since at some point of time, system cannot allocate more memory.

int\* func() {

int\* mem = malloc(1024);

return mem;

}

int\* mem = func(); /\* still accessible \*/

In the upper example, the allocated memory is still valid and accessible, even though the function terminated. When you are done with the memory, *you have to free it*:

free(mem);

