**Poznámka Autora:**

* Když pochopíte, co ty jednotlivý kódy dělají, tak je psát do c# nemusíte. Ony samy o sobě nic neukážou. Jde o to co se děje v paměti, když se executujou, což se asi nejlíp ukazuje, když je máte napsaný před sebou, ale to už je vaše věc.
* Ty obrázky se doporučuju naučit, je to taková věc, která se dobře okecává, navíc si jsem skoro jistý, že minimálně ten základní, kde proměnná ukazuje ze stacku na instanci na heapu (Příklad 4), po nás budou chtít.
* Hlavně si pamatujte, že string je na heapu
* To s tím linked listem má Mandík rád, takže je to dobrý taky umět.
* tahle stránka něco z toho dost dobře shrnuje, takže kdyby všechno nebylo jasný, tak tu dost možná najdete odpověď. http://clarkkromenaker.com/post/csharp-structs/

**Adresování a správa paměti, Garbage collector**

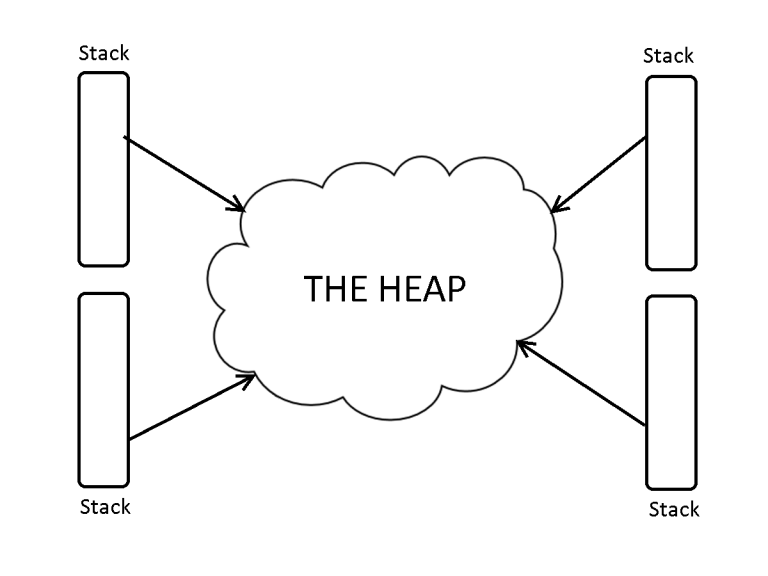
* Paměť v C# se rozděluje na 2 typy
  + Halda (heap)
  + Zásobník (Stack)

**Halda**

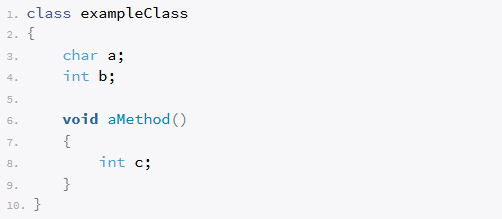
* Je pouze jedna v celém programu
* Na haldu se ukládají všechny referenční typy (třídy, interface, delegáti, objekty a stringy)
* Není, jakkoliv organizovaná
* Je přístupná pro všechny (pokud instanci nezablokujeme modifikátorem přístupu – public, private, internal, protected)
* Pokud jsou na haldě nějaká data, na které neukazuje žádná proměnná, smaže je garbage collector

**Zásobník**

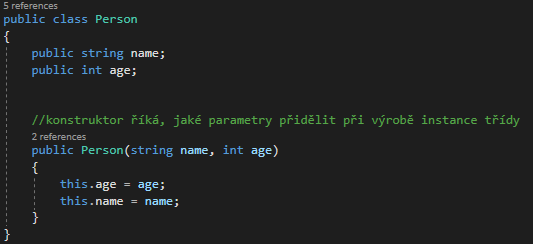
* Druh paměti, který vlákna využívají pro provedení kódu
* Každé vlákno má vlastní zásobník
* Je setříděný (LIFO)
* Každá metoda je vlastně blok, kde jsou uložena její data, bloky jsou vrstveny na sobě takovým způsobem, že nově přidané bloky padají na starší a tím zabraňují práci s těmi předešlými, dokud nejsou vyřešeny..
* Obsahuje pouze primitivní datové typy metod
* Proměnné volané v metodách, se také ukládají na zásobník, jsou ale nepřístupné, a jakmile metoda skončí, smažou se
* Pokud je práce s paměťovým blokem ukončena tak je i s daty smazán (Neni ve skutečnosti smazán, ale časem přepsán jiným blokem. Úplně stejné to funguje všude v PC)

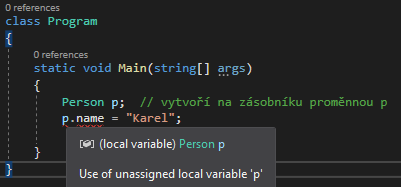


* Hodnotové typy (bool, byte, char, enum, float, int, long…) se uloží tam, kde jsou deklarovány



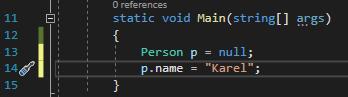
* V tomto kódu by proměnné a, b byly uloženy na haldě (protože to jsou proměnné třídy) a proměnná c by byla uložena na zásobníku (protože je to proměnná metody)

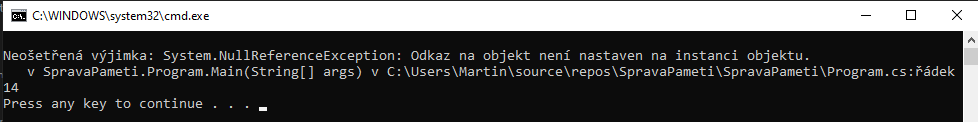
**Příklad 1:**



* V tomto příkladu program bude fungovat, dokud se nepokusíme pracovat s proměnnou p, poté nás kompilátor ani nenechá spustit kód
* Problémem je, že proměnná p neodkazuje na žádnou instanci na heapu

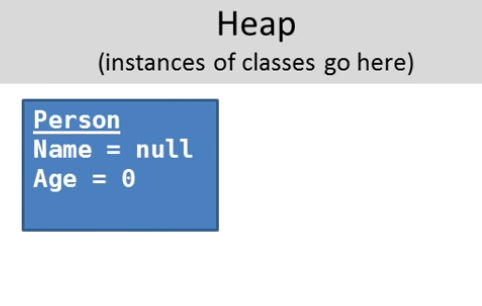
**Příklad 2:**



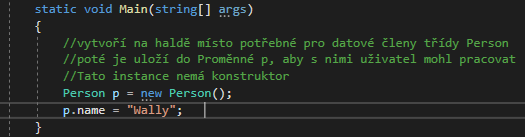
* Tento příklad dělá téměř to samé co ten předešlý
* Tentokrát však kompilátor kód zpracuje a zastaví ho až runtime, který vyhodí Null reference exception
* Hodnota null znamená, že referenční proměnná neukazuje na žádnou instanci třídy

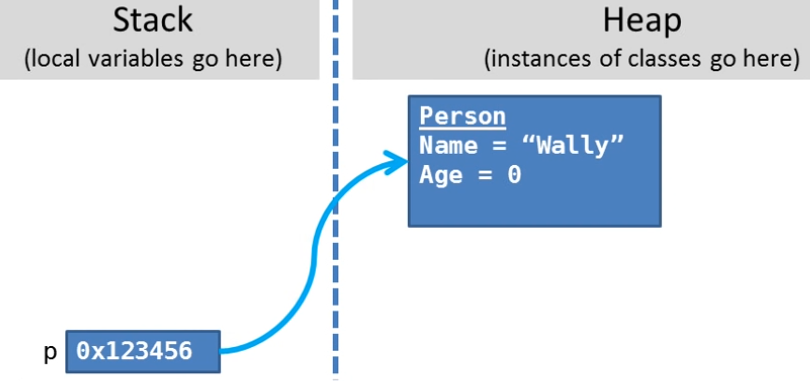
**Příklad 3**

* Pokud chceme vytvořit referenci na haldu, použijme klíčové slovo **new**
* Tento příkaz musí být uložen do nějaké proměnné, jinak ho smaže Garbage Collector
* Například příkaz **new Person();** by vytvořil na haldě místo potřebné pro datové členy třídy Person
* Takto by vypadala prázdná instance třídy Person



**Příklad 4**

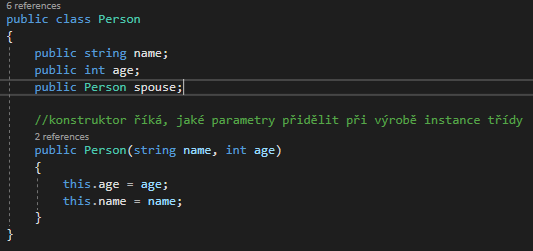
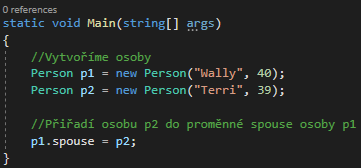
* Tento kód jednoduše vytvoří instanci třídy Person a uloží ji do proměnné p
* Poté se podívá, na jakou instanci referuje p a přepíše zde proměnnou name 



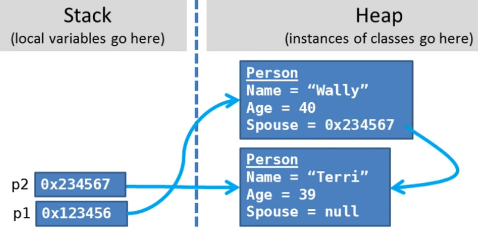
* V případě, že bychom měli napsaný konstruktor jako v příkladu 1 by kód vypadal takto:

**Příklad 5**

* Řekněme, že chceme k Person přidělit manželku (spouse), která je typu Person také

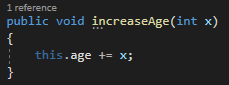
 

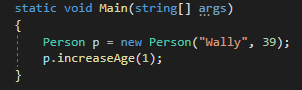
* Pokud bychom přiřadili Person p2 do proměnné spouse Person p1, stalo by se to, že by se adresa instance Person p2 jednoduše překopírovala do samotné proměnné spouse Person p1



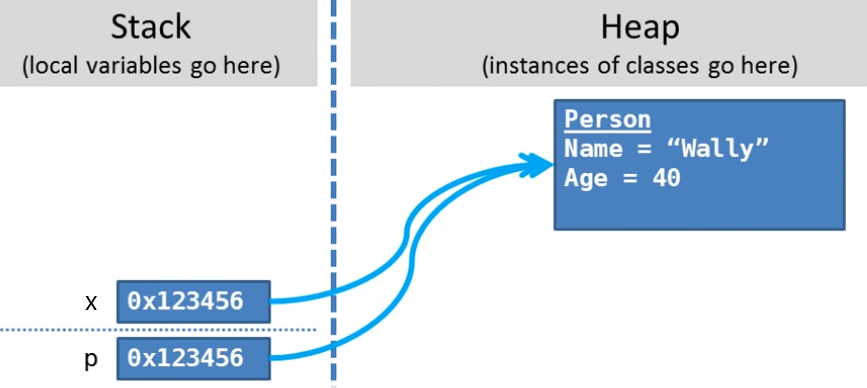
**Příklad 6**

* Tato metoda vezme věk osoby a přidá k němu hodnotu x





* Vytvoříme osobu a pomocí
* V paměti nám metoda increaseAge vytvoří proměnnou x na zásobníku. Ta se pak přičte k instanci dané osoby



**Stringy v paměti**

* Stringy nejsou hodnotový, ale referenční typ, to znamená, že jsou vždy uloženy na haldě.
* Když tedy například chceme změnit jméno Osoby

p.Name = “Karel“;

* Vytvoří se nový string na heapu apřepíše starou referenci**.**
* Starý string z haldy okamžitě nemizí, ale po chvíli bude odstraněn garbage collectorem

**Struktury**

* Struktury jsou podobné třídám, nemohou se však dědit nebo být děděny.
* Narozdíl od tříd jsou uloženy na stacku.
* Více o strukturách v otázce “Datové typy, Generika, Výčtové datové typy, Struktury, Delegáti, Eventy” od Karla Muroně.
* Taky je dobré zmínit, že všechny primitivní typy v C# jsou struktury, proto můžeme například napsat int. a zobrazí se nám různé metody. Funguje pouze v mainu, pokud to zkusíme v nějaké třídě, nic nám to neukáže, protože třída je na heapu a to znamená, že i struktura vytvořená ve třídě se uloží na heap.

**Dynamická pole v paměti**

* Všechna pole se ukládají na haldu (nezáleží jestli dynamická nebo statická)
* Při deklaraci mají nějakou určenou velikost.
* Uživatel k nevyužitým prvkům nemá přístup a ani je nevidí (může se ale podívat v debugu)
* Po tom co je velikost pole překonána vytvoří se na haldě nové, větší a překopírují se do něj prvky ze starého pole
* **Výjimkou je Linked list,** ten kvůli své řetězové architektuře nemá žádnou předem určenou velikost
* Díky tomu se může na haldu uložit, jak chce a nikdy nepřijde chvíle, kdyby se musel vytvářet znova za účelem zvýšení kapacity

**Proč indexujeme od 0?**

* Když načítáme prvek z pole, tak se vždy načte adresa prvního prvku (báze) a jeho velikost
* Pointer ze Stacku na něj ukazuje
* Všechny prvky jsou stejně velké
* Číslo v závorkách poté znamená o kolik se od tohoto načteného prvku má program posunout(offsetnout)
* To znamená, že pokud máme například toto pole:



* A řekněme, že chceme první prvek
* Tím pádem si musíme říct: “máme první prvek a o kolik míst se musíme posunout od prvku, který už máme načtený?”(kolikrát se posuneme o velikost prvního prvku) O nula míst. Stejně to vypadá i v kódu:



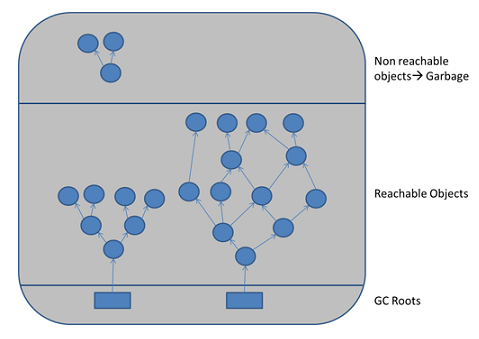
* Kdybychom chtěli třeba 3. prvek tak je postup úplně stejný
* Máme načtený první prvek a o kolik míst se musíme posunout, aby jsme dostali 3 prvek. O 2 místa (2x velikost prvního prvku)



* Program se posouvá právě o velikost prvního prvku. Proto je procházení listu tak rychlé, protože nemusí procházet prvek za prvkem, ale pouze se posune o urćenou vzdálenost na potřebné místo, kde teprve načte prvek. Neplatí například v Linked listu, kde kvůli jeho architektuŕe musí program projít každým jednotlivým prvkem.

**Garbage Collector**

* Prochází haldu a maže objekty, které nemají referenci na žádnou proměnnou
* „Živé“ objekty si vkládá do grafu, který je podobný stromu (strom to ale není, cyklí se), ty které v grafu nejsou, jsou označeny jako odpad ke smazání
* Rozděluje paměť v haldě do 3 kategorií Generation 0, Generation 1 a Generation 2



Generation 0

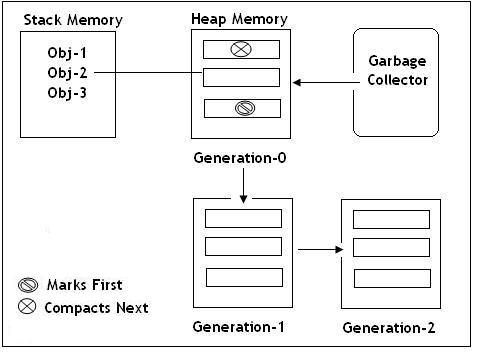
* Místo, kde se nacházejí právě vytvořené objekty (objekty s krátkou životností)
* Pokud je místo v Generation 0 už plné garbage collector ji pročistí, a ty které nemaže, jsou přesunuty do Generation 1
* Toto místo je čištěno nejčastěji
* Garbage collector se spouští automaticky, ale může být také zavolán uživatelem … System.GC.Collect()

Generation 1

* Vše se opakuje stejně jako u Generation 0
* Objekty, které nejsou smazané, se přesunou do Generation 2

Generation 2

* Místo, kde garbage collector zasahuje ze všech nejméně
* Nacházejí se zde pouze objekty s dlouhou životností, které často už potom vydrží po celou dobu chodu programu
* Jedná se například o objekty vytvořené pro statické členy nebo globální proměnné



Object Generation

For better performance of memory releasing, the managed heap is divided into segments called “Generations”. There are only 3 generations: 0, 1 and 2. This idea of generations is simple: the longer an object has existed on the heap are reachable objects. Objects that have only recently been placed on the heap are unreachable.

1. When objects are just created, they are placed to the Generation 0 (Gen 0).
2. When Gen 0 is full, the GC performs a garbage collection. During the collection, the GC removes all unreachable objects from the heap. All reachable objects are promoted to the Generation 1 (Gen 1). The Gen 0 collection is a rather quick operation.
3. When Gen 1 is full, the Gen 1 garbage collection is performed. All objects that survive the collection are promoted to Gen 2. The Gen 0 collection also takes place here.
4. When Gen 2 is full, the GC performs full garbage collection. First, Gen 2 collection is performed, then the Gen 1 and Gen 0 collections take place. If there is still not enough memory for new allocations, the GC raises the OutOfMemory exception.
5. During a full garbage collection, the GC must pass through all objects in the heap, so this process might have a great impact on system resources.

There are no specific timings for GC to get triggered, GC automatically starts operation on the following conditions:

When virtual memory is running out of space.

When allocated memory is suppressed acceptable threshold (when GC found if the survival rate (living objects) is high, then it increases the threshold allocation).

When we call GC.Collect() method explicitly, as GC runs continuously, we actually do not need to call this method.

**Zdroje:**

[**https://www.c-sharpcorner.com/article/C-Sharp-heaping-vs-stacking-in-net-part-i/**](https://www.c-sharpcorner.com/article/C-Sharp-heaping-vs-stacking-in-net-part-i/)

[**https://dev.to/tyrrrz/interview-question-heap-vs-stack-c-5aae**](https://dev.to/tyrrrz/interview-question-heap-vs-stack-c-5aae)

[**https://www.loadingdeveloper.com/heaps-and-stacks-in-csharp/**](https://www.loadingdeveloper.com/heaps-and-stacks-in-csharp/)

[**https://www.youtube.com/watch?v=clOUdVDDzIM**](https://www.youtube.com/watch?v=clOUdVDDzIM)