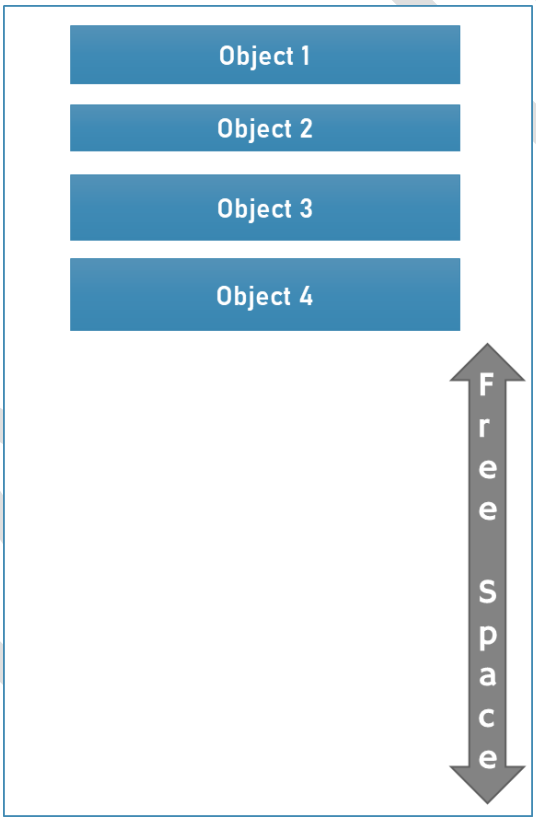
33

Garbage Collection, Destructor, IDisposable

Introducing Garbage Collection

› Garbage Collection is a process of deleting objects from memory, to free-up memory; so the same memory can be re-used.

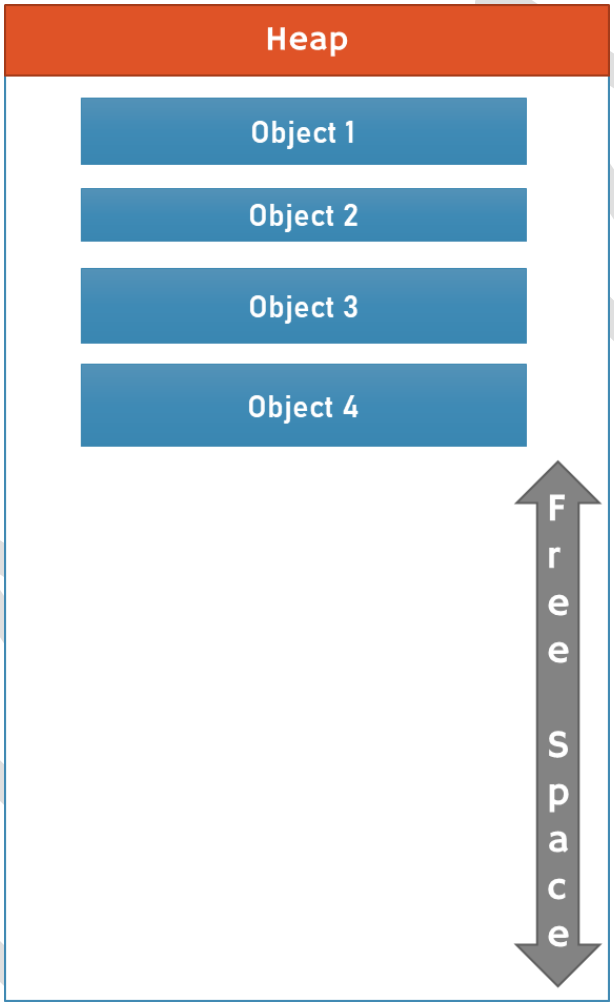


How Garbage Collection Works?

› CLR automatically allocates memory for all objects created any where in the application, whenever it encounters "new ClassName( )" statement. This process is called as "Memory Management", which is done by "Memory Manager" component of CLR.

› All objects are stored in "Heap" (a.k.a. virtual memory).

- Heap is only-one for the entire application life time.
- The default heap size 64 MB (approx.), and extendable.
- When CLR can't find space for storing new objects, it performs a process called "Garbage Collection" automatically, which includes "identification of un-referenced objects and deleting them from heap; so that making room for new objects". This process is done by "Garbage Collector (GC)" component of CLR.

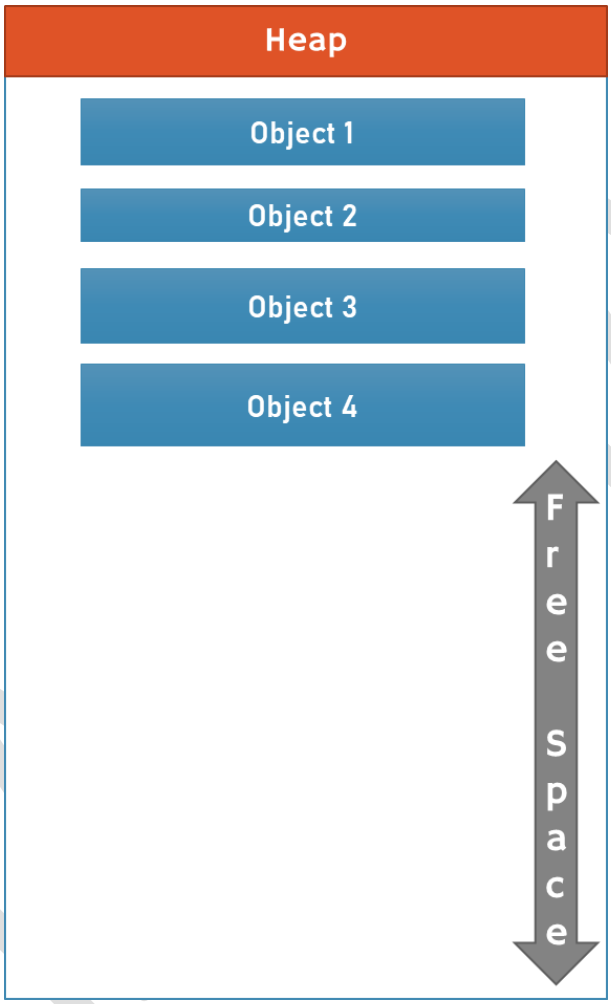


How GC decides if objects are alive?

- GC checks belongs information from the MSIL code:
  - It collects references of an object.

- It identifies whether any object is referenced by static field.

- The objects that has at least one living reference variable in any stack or static field, are "alive objects"; others are "dead objects" or "un-used objects".



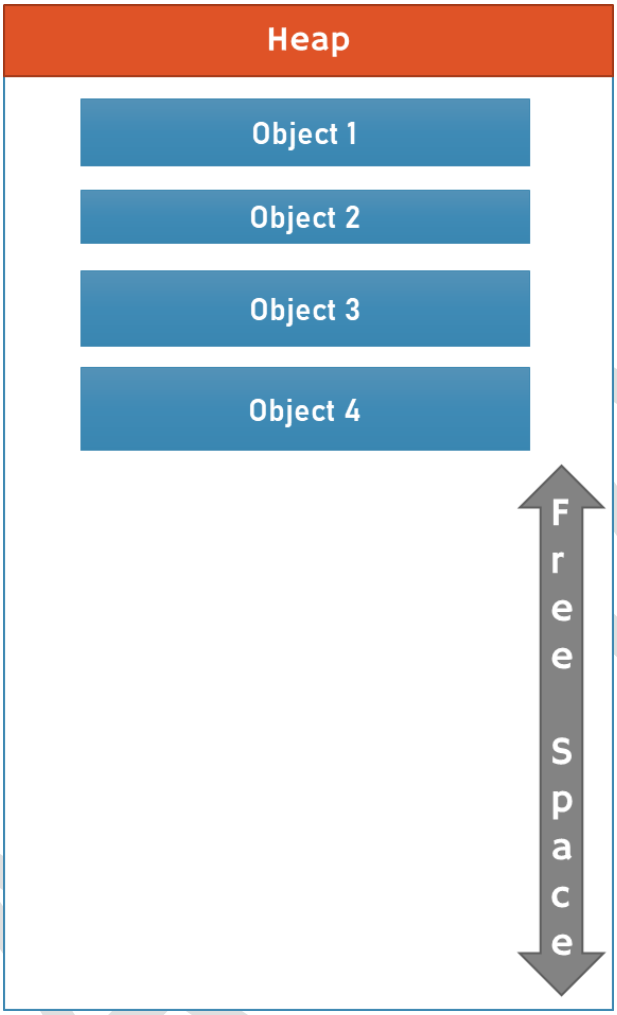
When GC gets triggered?

- There are NO specific timings for GC to get triggered.

- GC automatically gets trigged in the following conditions:

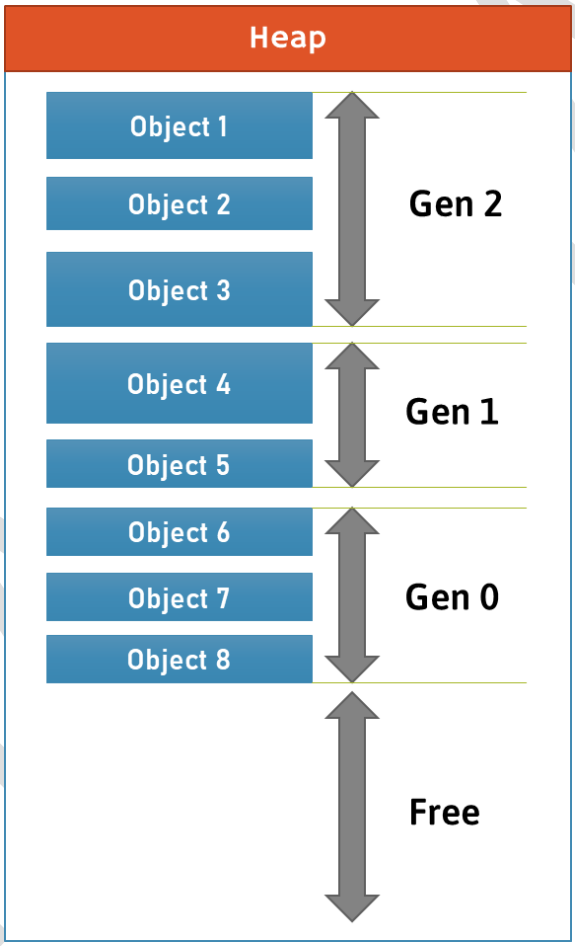
  - When the "heap" is full or free space is too low.

› When we call GC.Collect() explicitly.



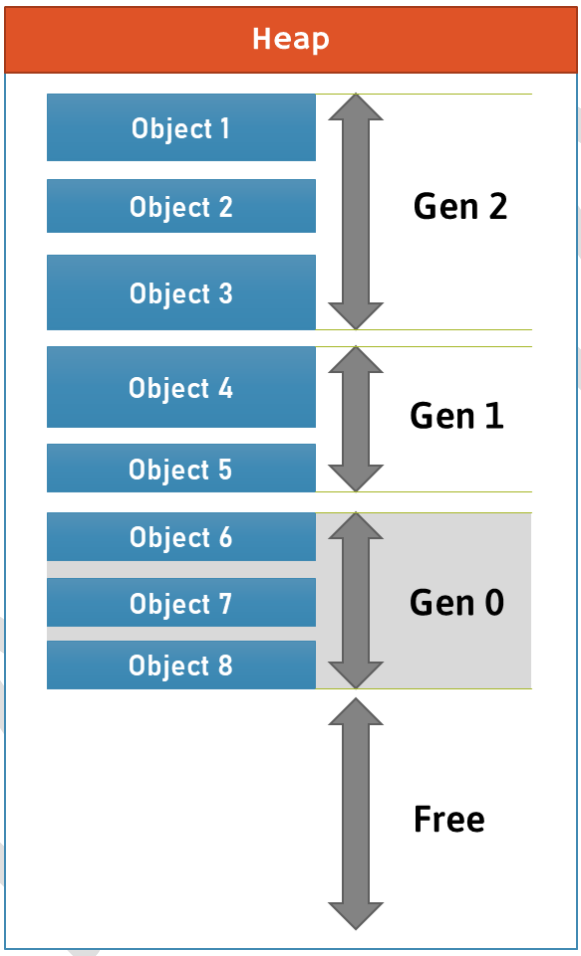
Generics in GC

- Heap contains three segments (called generations):
  - Generation 2 [Long-Lived Generation]
  - Generation 1 [Survival Generation]
  - Generation 0 [Short-Lived Generation]



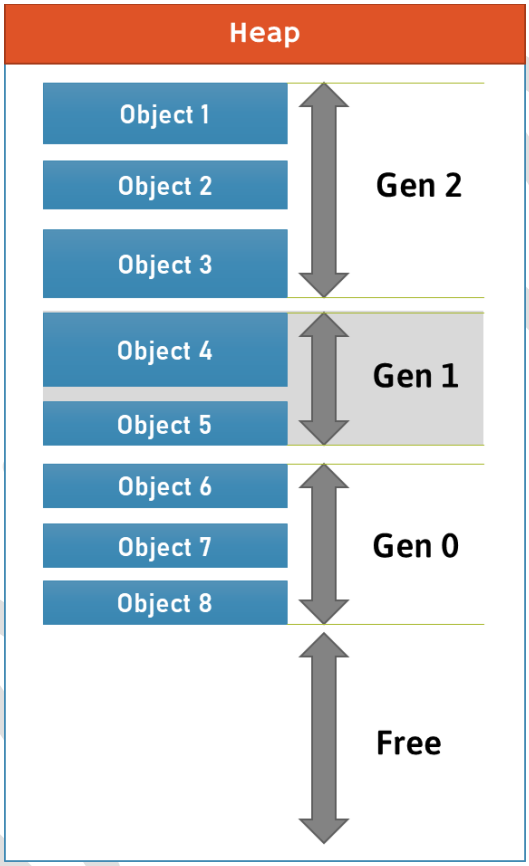
Generation 0

- The "Generation 0" is the youngest generation and contains newly created short-lived objects and collected at first priority. The objects survive longer, are promoted to "Generation 1".
- Ex: The newly created objects.



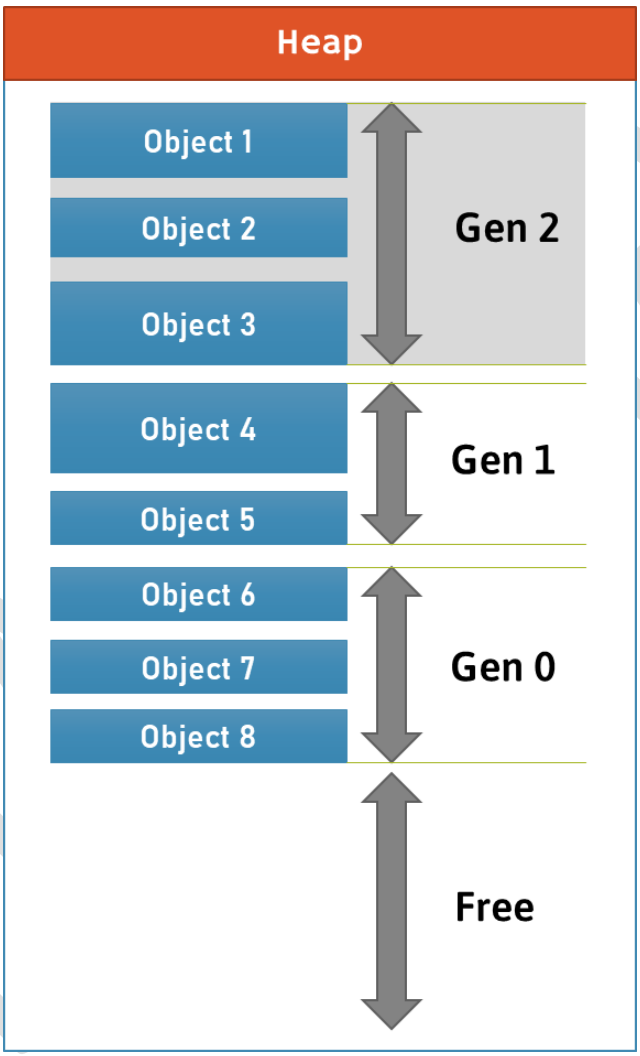
Generation 1

- The "Generation 1" is buffer between "Generation 0" and "Generation 2".
- The "Generation 1" mainly contains frequently-used and longer-lived objects.
- Ex: The objects created in the previously-executed methods, but still accessible.



Generation 2

- The "Generation 2" contains the longest-lived objects that were created long-back and still is being used, by different statements in the program.
- Ex: The objects that referenced with static fields.



Managed (vs) Unmanaged Resources

Managed Resources

- The objects that are created by CLR are called as "Managed Resources".

- These will participate in "Garbage Collection" process, which is a part of .NET Framework.

Unmanaged Resources

- The objects that are not created by CLR and not managed by CLR are called as "Un-managed resources".
- Ex: File streams, database connections

Clearing Unmanaged Resources

Destructor

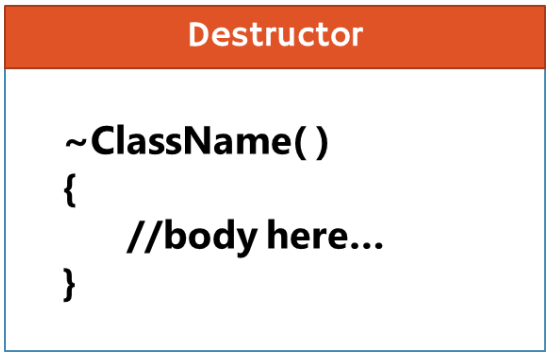
- Clears unmanaged resources just before deleting the object; i.e. generally at the end of application execution.

Dispose

- Clears unmanaged resources after the specific task (work) is completed; so no need to wait till end of application execution.

Introducing Destructor

- Destructor is a special method of the class, which is used to close un-managed resources (such as database connections and file connections), that are opened during the class execution.



- **Advantage:** We close database connections and file connections; so no memory wastage or leakage.
- Destructor doesn't de-allocate any memory; it just will be called by CLR (.net runtime engine) automatically, just before a moment of deleting the object of the class.
- Destructor's name should be same as class name, started with ~ (tilde) character.
- A Destructor is unique to its class i.e. there cannot be more than one destructor in a class.
- Destructor can't have parameters or return value.
- Destructor is "public" by default, we can't change its access modifier.
- Destructor doesn't support any other modifiers such as "virtual", "abstract", "override" etc.
- Destructors can be defined only in classes; but not in structs, interfaces etc.
- Destructors can't be overloaded or inherited.
- Destructors are usually called at the end of program execution.

Destructor

```
~ClassName
{
    //body here...
}
```

compile

Destructor [After compilation]

```
protected override void Finalize()
{
    try
    {
        //code of destructor
    }
    catch
    {
        //code of destructor
    }
    finally
    {
        base.Finalize();
    }
}
```

Destructor (vs) Finalize Method

- Internally, destructor is compiled as the "Finalize" method.
- The "destructor" is a term belongs to C# language; the "Finalize" method belongs to .net framework generally; and both are same (interchangeable).
- The compiled Finalize method calls the Finalize method of corresponding base class.

Introducing IDisposable and Dispose

- The "IDisposable" interface of "System" namespace, has a method called "Dispose", which is used to close un-managed resources that are created during the life-time of the object.

Implementing System.IDisposable interface

```
class ClassName : System.IDisposable
{
    public void Dispose()
    {
        //Close un-managed resources here
    }
}
```

Creating object with IDisposable

```
using (ClassName referenceVariable = new ClassName() )
{
    //your code here
}
```

- The un-managed resources include file streams and database connections.
- At the end of "using" statement, automatically "Dispose" method will be called.
- Dispose is better than Destructor, because we need wait till 'end of application execution' to clear unmanaged resources; we clear them immediately after usage.

'Using' Declaration

- You can prefix "using" keyword before the local variable declaration, in order to call "Dispose" method when that variable goes out of scope.
- New feature introduced in C# 8.0

Creating object

```
public void Method()
{
    using ClassName referenceVariable = new ClassName();

    //do work here

} //Dispose will be called automatically here
```