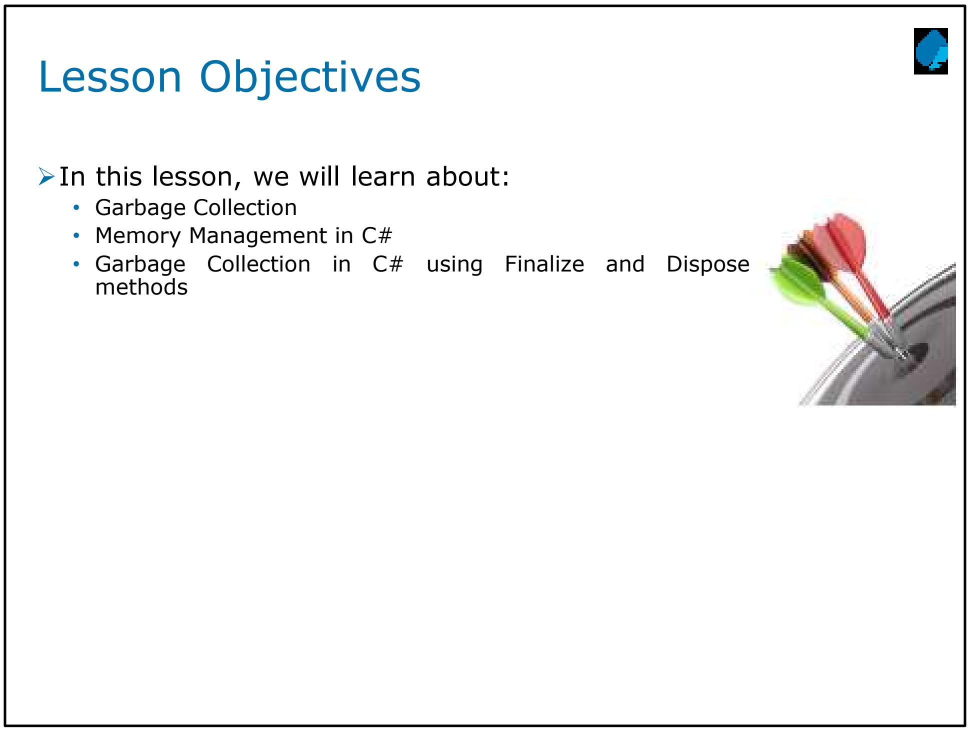
.NET Framework 4.6 and C# 7.0

Lesson 08 : Garbage Collection

Capgemini

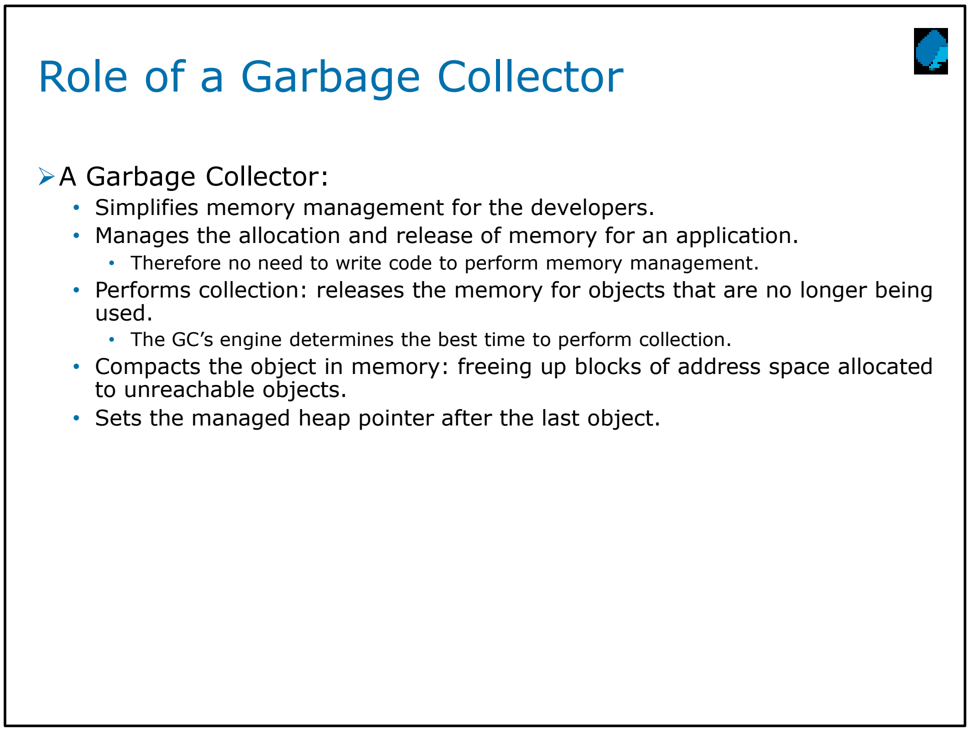
Explain the lesson coverage



Lesson Objectives

- In this lesson, we will learn about:
  - Garbage Collection
  - Memory Management in C#
  - Garbage Collection in C# using Finalize and Dispose methods

C#'s garbage collection system reclaims objects automatically—occurring transparently, behind the scenes, without any programmer intervention. It works like this: When no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object is released. This recycled memory can then be used for a subsequent allocation.

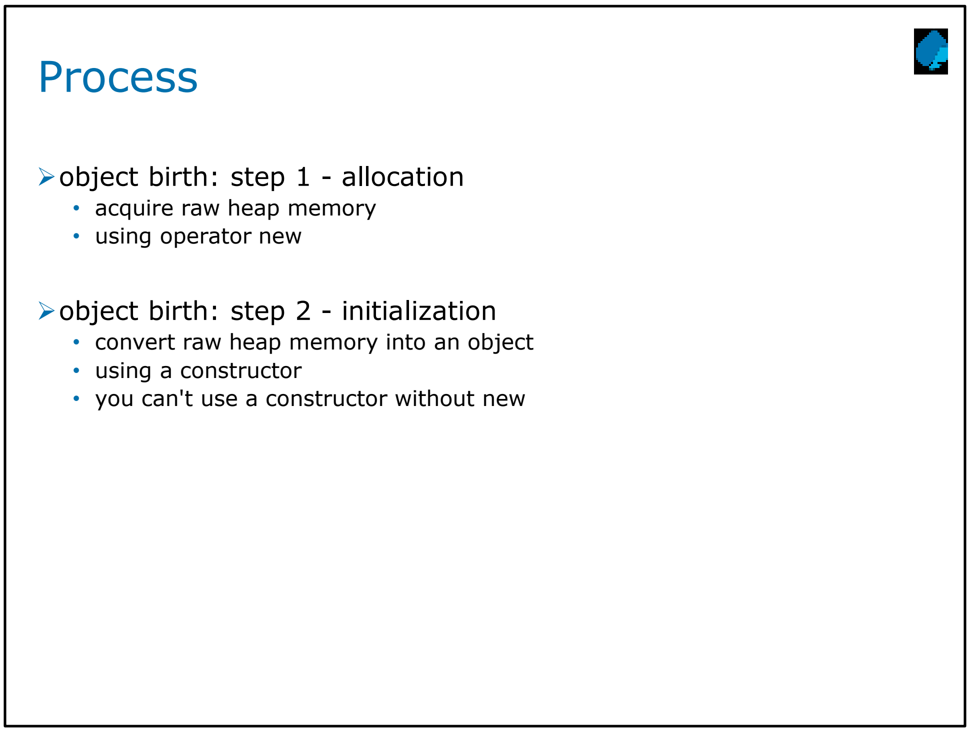


Role of a Garbage Collector

- A Garbage Collector:
  - Simplifies memory management for the developers.
  - Manages the allocation and release of memory for an application.
    - Therefore no need to write code to perform memory management.
  - Performs collection: releases the memory for objects that are no longer being used.
    - The GC's engine determines the best time to perform collection.
  - Compacts the object in memory: freeing up blocks of address space allocated to unreachable objects.
  - Sets the managed heap pointer after the last object.

Roles of a Garbage Collector:
As you have seen, objects are dynamically allocated from a pool of free memory by using the new operator. Of course, memory is not infinite, and the free memory can be exhausted. Thus, it is possible for new to fail because there is insufficient free memory to create the desired object. For this reason, one of the key components of any dynamic allocation scheme is the recovery of free memory from unused objects, making that memory available for subsequent reallocation. In many programming languages, the release of previously allocated memory is handled manually. For example, in C++, you use the delete operator to free memory that was allocated. However, C# uses a different, more trouble-free approach: garbage collection.

The new operator has a high precedence (just below ++ and --), and cannot be overloaded in C# (unlike in C++). Thus in C# memory allocation is completely automatic and cannot be controlled in any way.



Process

- object birth: step 1 - allocation
  - acquire raw heap memory
  - using operator new
- object birth: step 2 - initialization
  - convert raw heap memory into an object
  - using a constructor
  - you can't use a constructor without new

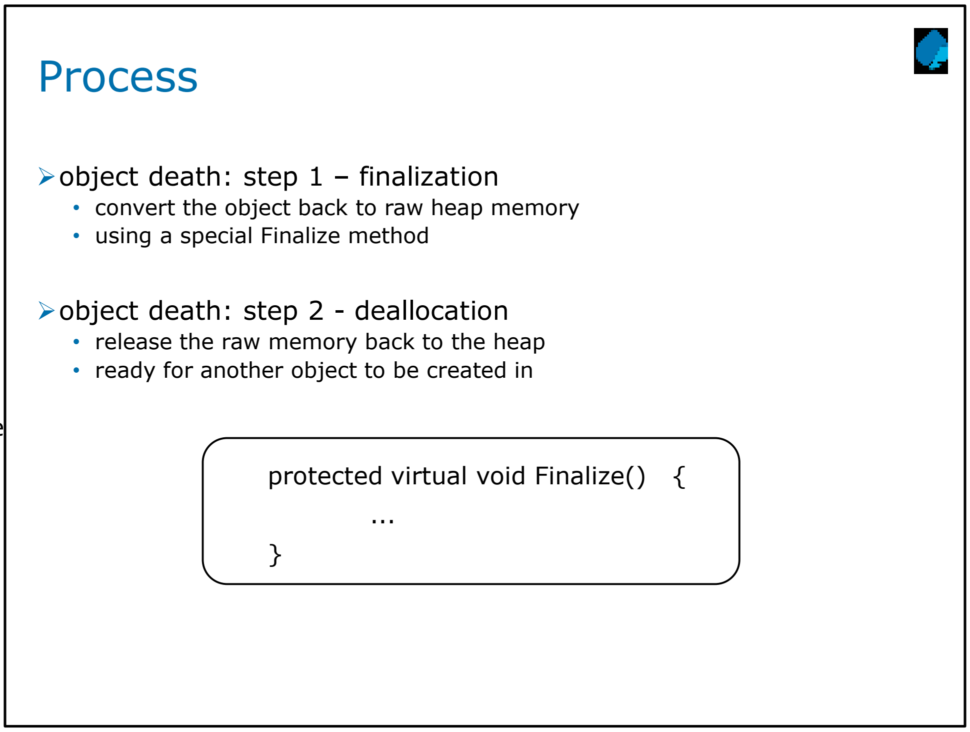
Creating an Object:
Creating an object is a two phase process.

The first step is to acquire some raw memory that the object is to be created in. This is called allocation. The memory for a value is automatically allocated from the stack. However, this chapter is specifically only considering objects. The memory for an object is allocated from the heap using the new operator. If the allocation fails because heap memory is exhausted, the runtime will throw an OutOfMemoryException.

The second step is to convert the raw memory into an object. This is called initialization. The only way to initialize an object is via a constructor. You can write a constructor to control the steps taken to initialize an object.

Thus in C# you control when an object is created by writing a new expression, which combines the new operator and a call to a constructor and you also control the steps taken to construct the object in the raw memory. The only step you do not and cannot control is the allocation of the memory.

Thus destroying an object in C# is the exact mirror of creating an object. You can control the steps taken to finalize the object (by overriding Object.Finalize) just as you can control the steps taken to initialize an object (by writing a constructor).



Process

- object death: step 1 – finalization
  - convert the object back to raw heap memory
  - using a special Finalize method
- object death: step 2 - deallocation
  - release the raw memory back to the heap
  - ready for another object to be created in

```
protected virtual void Finalize()   {
            ...
}
```

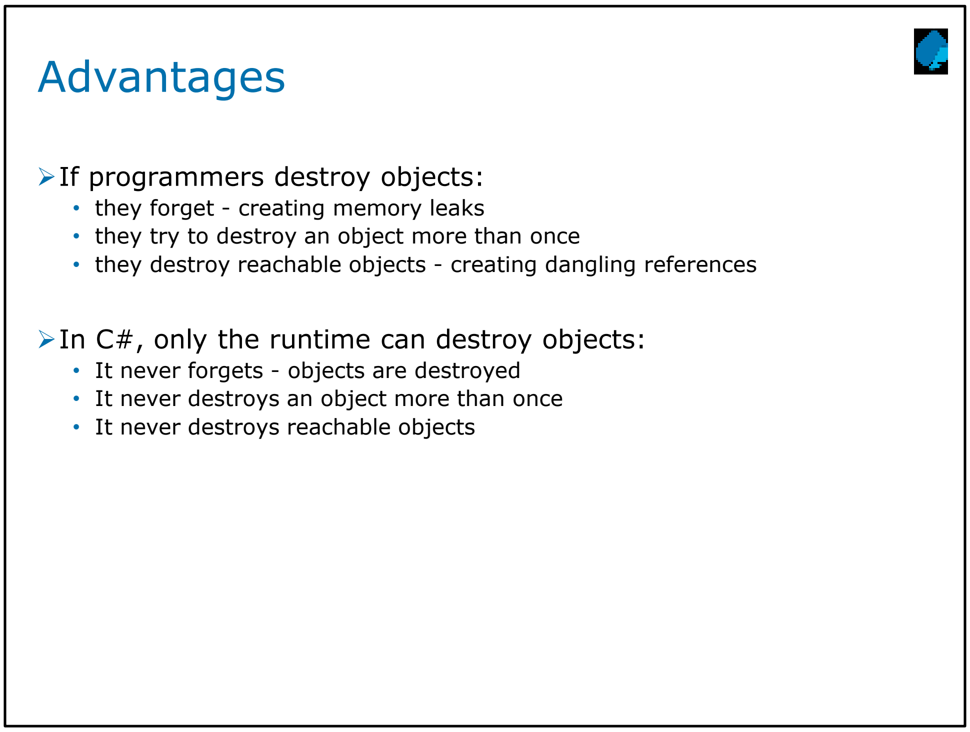
Destroying an Object:

Destroying an object is also a two phase process.

First, the object has to be converted back to raw heap memory. This is called finalization. C# finalizes an object by calling its Finalize method. All objects have a Finalize method because all objects implicitly derive from System.Object which contains Finalize as shown on the slide above. This means that a programmer can override Finalize in their class and control the steps that will be taken when the object is finalized.

Second, the raw memory is released back to the heap. This happens automatically and cannot be controlled by the programmer in any way.

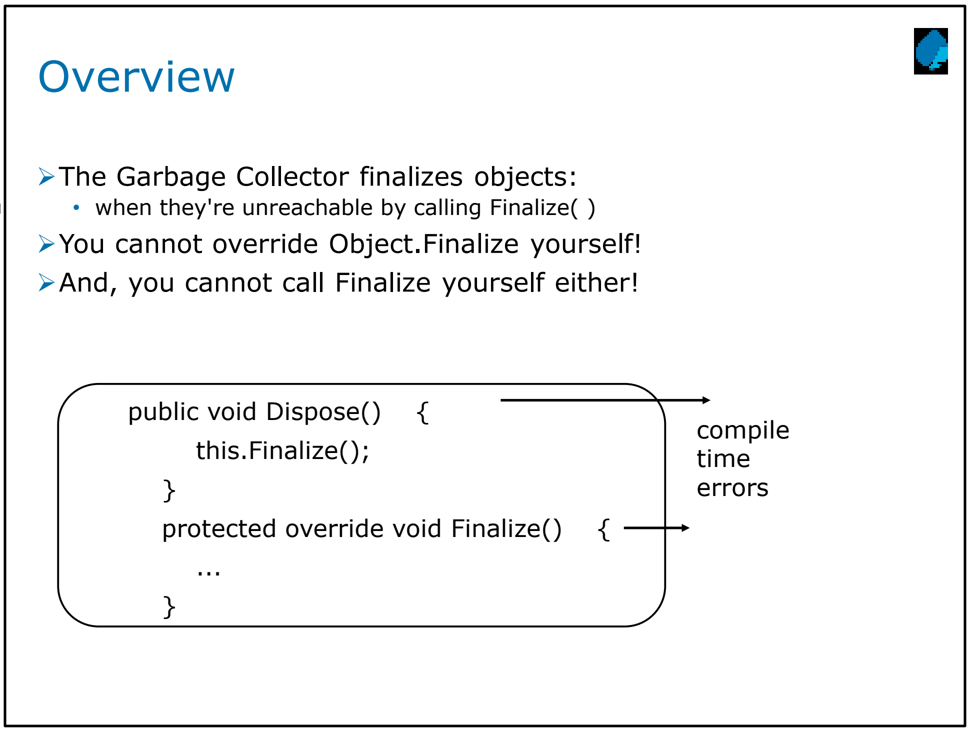
You cannot control the steps taken to deallocate an objects memory just as you cannot control the steps taken to allocate the memory in the first place.



Advantages

- If programmers destroy objects:
  - they forget - creating memory leaks
  - they try to destroy an object more than once
  - they destroy reachable objects - creating dangling references
- In C#, only the runtime can destroy objects:
  - It never forgets - objects are destroyed
  - It never destroys an object more than once
  - It never destroys reachable objects

Advantages of Destroying an Object:
Programmers can create objects in code by using new expressions at any time they wish. However, programmers cannot destroy objects in code. The reasons for this major design decision are given in the slide above; if the programmer is responsible for destroying objects then they'll sometimes forget to destroy the object. As the slide says, this causes a memory leak since the memory is not returned to the heap. Perhaps worse is that any finalisation steps are not run. They'll try to destroy an object more than once causing untold chaos.
For these reasons, C# does not allow the programmer to destroy objects. It is completely impossible. Instead the C# runtime destroys objects for you. The part of the C# runtime that has this responsibility is called the garbage collector. The garbage collector ensures that:
Objects are destroyed, even if it's only when the program shuts down. Thus you can be sure that critical finalization code will run.
Objects are not destroyed more than once. Thus garbage collector always maintains its own data structures in a consistent state.
Objects are not destroyed while they are still reachable. To guarantee this the garbage collector maintains a graph of all reachable objects.



Overview

- The Garbage Collector finalizes objects:
  - when they're unreachable by calling Finalize( )
- You cannot override Object.Finalize yourself!
- And, you cannot call Finalize yourself either!

```
public void Dispose()    {
      this.Finalize();
   }
   protected override void Finalize()    {
      ...
   }
```

compile
time
errors

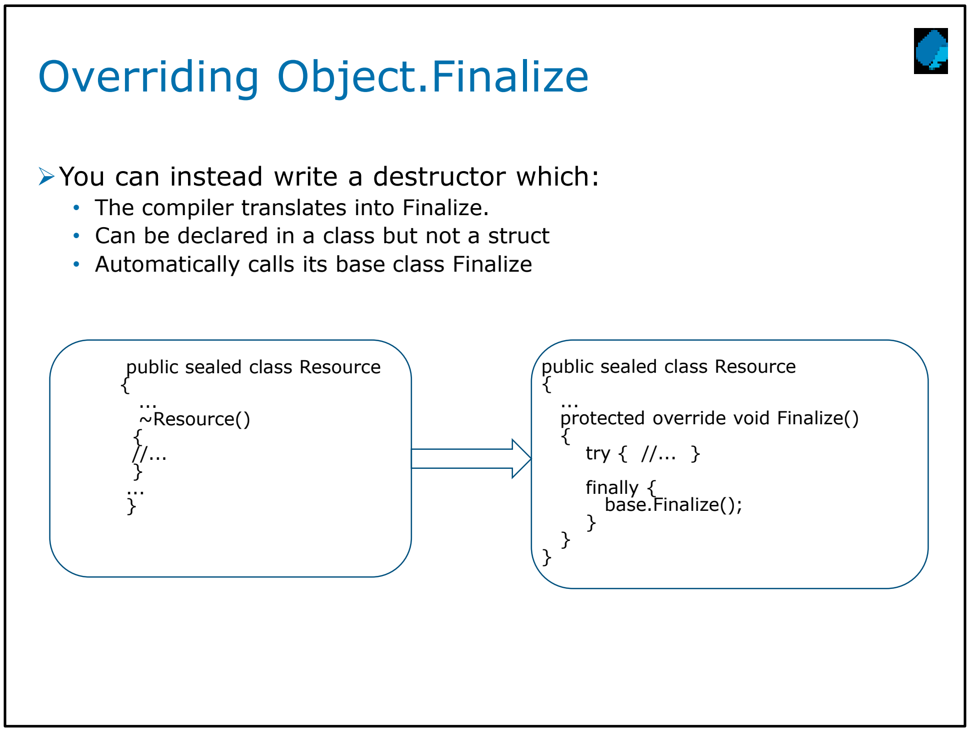
Note that while a Dispose method cannot call Finalize, a Finalize method can call Dispose making it easy to call Dispose twice, once by the programmer and once by the garbage collector. In these Cases the programmer must ensure that multiple calls to Dispose are benign.

Finalizers in Garbage Collection:

As you have seen, when an object becomes unreachable it is collected by the garbage collector. If the object overrides the Object.Finalize method then this method will be called by the garbage collector. However, you cannot directly override Object.Finalize. Despite the implicit inheritance any attempt to declare a Finalize override will result in a compile-time error. The reason that you cannot override Finalize explicitly is because if you could you would then be able to call Finalize explicitly in code (or there would have to be a messy special rule just for Finalize overrides). The reason that calling Finalize is not allowed is because if you could call Finalize you would be back to the inherent problems of double destruction; you'd call Finalize and then the garbage collector would call Finalize again.
For these reasons you cannot explicit override Object.Finalize and you cannot explicitly call Finalize. However, you can implicitly override Object.Finalize by writing a destructor.
Note: Finalize is a protected method. This means that a struct cannot override the Finalize method (structs can't declare protected members because you can't derive from a struct). However, this is okay because you would not want

to override Finalize in a struct anyway. Finalization is only for objects that live on the heap. It is not for values that live on the stack.

You can see this destructor-to-Finalize conversion in the IL code.



Overriding Object.Finalize

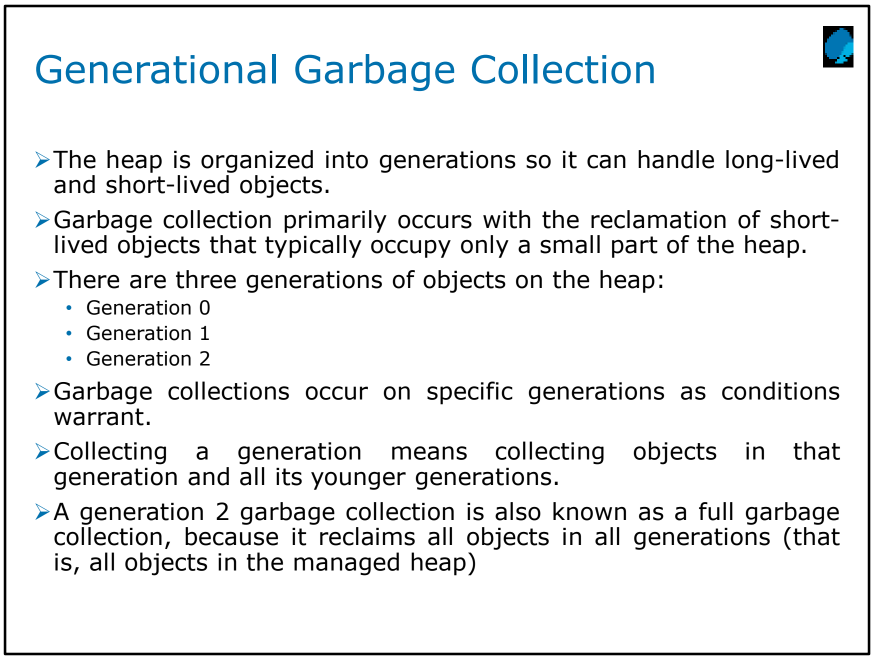
- You can instead write a destructor which:
  - The compiler translates into Finalize.
  - Can be declared in a class but not a struct
  - Automatically calls its base class Finalize

```
public sealed class Resource
{
   ...
   ~Resource()
   {
   //...
   }
  ...
}
```

```
public sealed class Resource
{
   ...
   protected override void Finalize()
   {
      try {  //...  }

      finally {
         base.Finalize();
      }
   }
}
```

Overriding Object.Finalize:
The way to create an override of Object.Finalize is to write a destructor. The C# compiler will automatically convert a destructor into an override of Object.Finalize. The syntax for declaring a destructor is to write a tilde (~) followed by the name of the class. A destructor never has a return type and never has any parameters.
The reason that a destructor has no return type and no parameters is that you cannot call the destructor. Remember, the destructor is really just the Finalize method in disguise. (Notice that destructors and Finalize both have no return type and no parameters.)
The destructor is converted into an override of Object.Finalize in a very special way. The statements inside the destructor are automatically placed inside a try block inside the override. The try block is followed by a finally block in which the base class Finalize is explicitly called. This is only legal because this call is being done in code that is generated by the C# compiler and will only be run by the garbage collector.



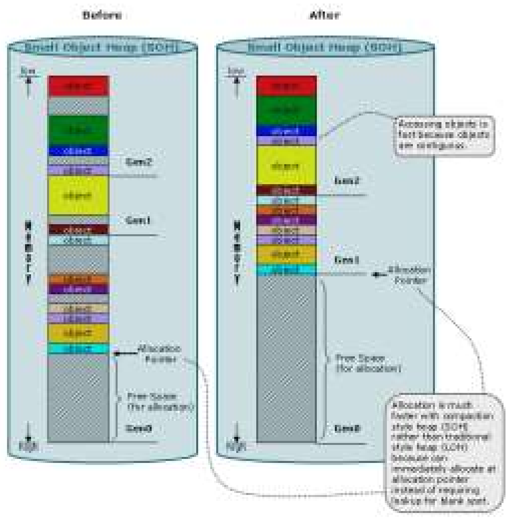
Generational Garbage Collection

- The heap is organized into generations so it can handle long-lived and short-lived objects.
- Garbage collection primarily occurs with the reclamation of short-lived objects that typically occupy only a small part of the heap.
- There are three generations of objects on the heap:
  - Generation 0
  - Generation 1
  - Generation 2
- Garbage collections occur on specific generations as conditions warrant.
- Collecting a generation means collecting objects in that generation and all its younger generations.
- A generation 2 garbage collection is also known as a full garbage collection, because it reclaims all objects in all generations (that is, all objects in the managed heap)

There are three generations of objects on the heap:

- **Generation 0**. This is the youngest generation and contains short-lived objects. An example of a short-lived object is a temporary variable. Garbage collection occurs most frequently in this generation. Newly allocated objects form a new generation of objects and are implicitly generation 0 collections, unless they are large objects, in which case they go on the large object heap in a generation 2 collection. Most objects are reclaimed for garbage collection in generation 0 and do not survive to the next generation.
- **Generation 1**. This generation contains short-lived objects and serves as a buffer between short-lived objects and long-lived objects.
- **Generation 2**. This generation contains long-lived objects. An example of a long-lived object is an object in a server application that contains static data that is live for the duration of the process.

Generational Garbage Collection

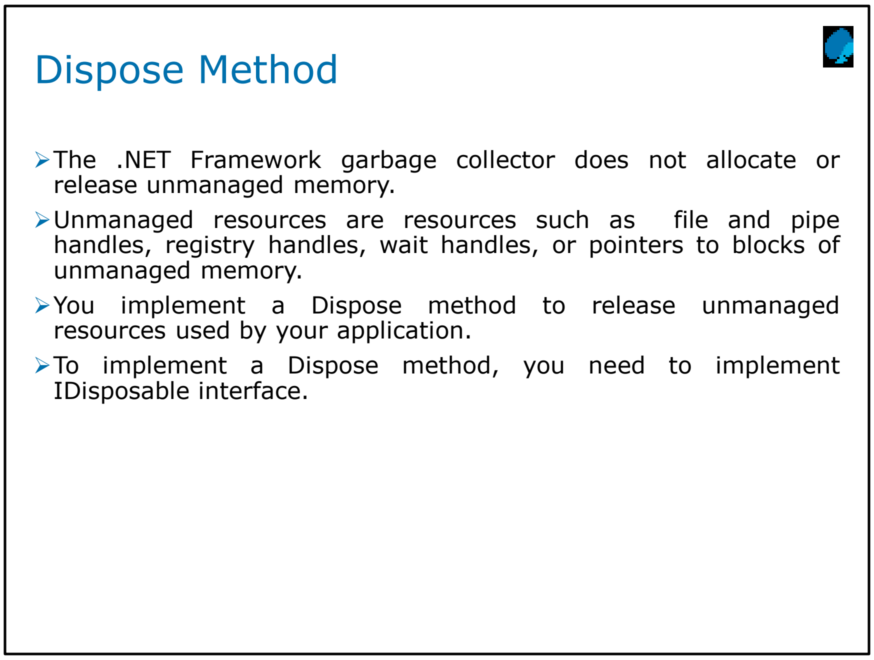

Before
After
Small Object Heap (SOH)
Small Object Heap (SOH)
Low
Memory
High
object
Gen2
Gen1
Gen0
Allocation Pointer
Free Space (for allocation)
Accessing objects is fast because objects are contiguous.
Allocation is much faster with compaction style heap (SOH) rather than traditional style heap (LOH) because can immediately allocate at allocation pointer instead of requiring lookup for blank spot.

Objects that are not reclaimed in a garbage collection are known as survivors, and are promoted to the next generation. Objects that survive a generation 0 garbage collection are promoted to generation 1; objects that survive a generation 1 garbage collection are promoted to generation 2; and objects that survive a generation 2 garbage collection remain in generation 2.
When the garbage collector detects that the survival rate is high in a generation, it increases the threshold of allocations for that generation, so the next collection gets a substantial size of reclaimed memory. The CLR continually balances two priorities: not letting an application's working set get too big and not letting the garbage collection take too much time.

A garbage collection has the following phases:

- A marking phase that finds and creates a list of all live objects.
- A relocating phase that updates the references to the objects that will be compacted.
- A compacting phase that reclaims the space occupied by the dead objects and compacts the surviving objects. The compacting phase moves objects that have survived a garbage collection toward the older end of the segment.
- Because generation 2 collections can occupy multiple segments, objects that are promoted into generation 2 can be moved into an older segment. Both generation 1 and generation 2 survivors can be moved to a different

segment, because they are promoted to generation 2.



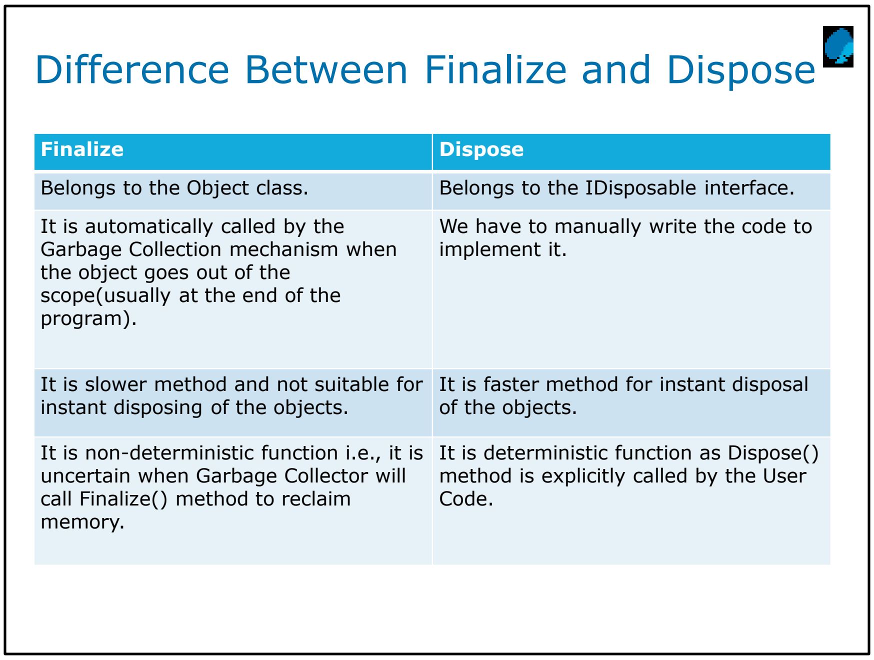
Dispose Method

- The .NET Framework garbage collector does not allocate or release unmanaged memory.
- Unmanaged resources are resources such as file and pipe handles, registry handles, wait handles, or pointers to blocks of unmanaged memory.
- You implement a Dispose method to release unmanaged resources used by your application.
- To implement a Dispose method, you need to implement IDisposable interface.

Dispose Method:

Class instances often encapsulate control over resources that are not managed by the runtime, such as window handles (HWND), database connections, and so on. Therefore, you should provide both an explicit and an implicit way to free those resources. Provide implicit control by implementing the protected Finalize on an object (destructor syntax in C# and C++). The garbage collector calls this method at some point after there are no longer any valid references to the object.

In some cases, you might want to provide programmers using an object with the ability to explicitly release these external resources before the garbage collector frees the object. If an external resource is scarce or expensive, better performance can be achieved if the programmer explicitly releases resources when they are no longer being used. To provide explicit control, implement the Dispose method provided by the IDisposable interface. The consumer of the object should call this method when it is finished using the object. Dispose() can be called even if other references to the object are alive.



Difference Between Finalize and Dispose

Finalize	Dispose
Belongs to the Object class.	Belongs to the IDisposable interface.
It is automatically called by the Garbage Collection mechanism when the object goes out of the scope(usually at the end of the program).	We have to manually write the code to implement it.
It is slower method and not suitable for instant disposing of the objects.	It is faster method for instant disposal of the objects.
It is non-deterministic function i.e., it is uncertain when Garbage Collector will call Finalize() method to reclaim memory.	It is deterministic function as Dispose() method is explicitly called by the User Code.

Difference between Finalize() and Dispose() methods in .NET

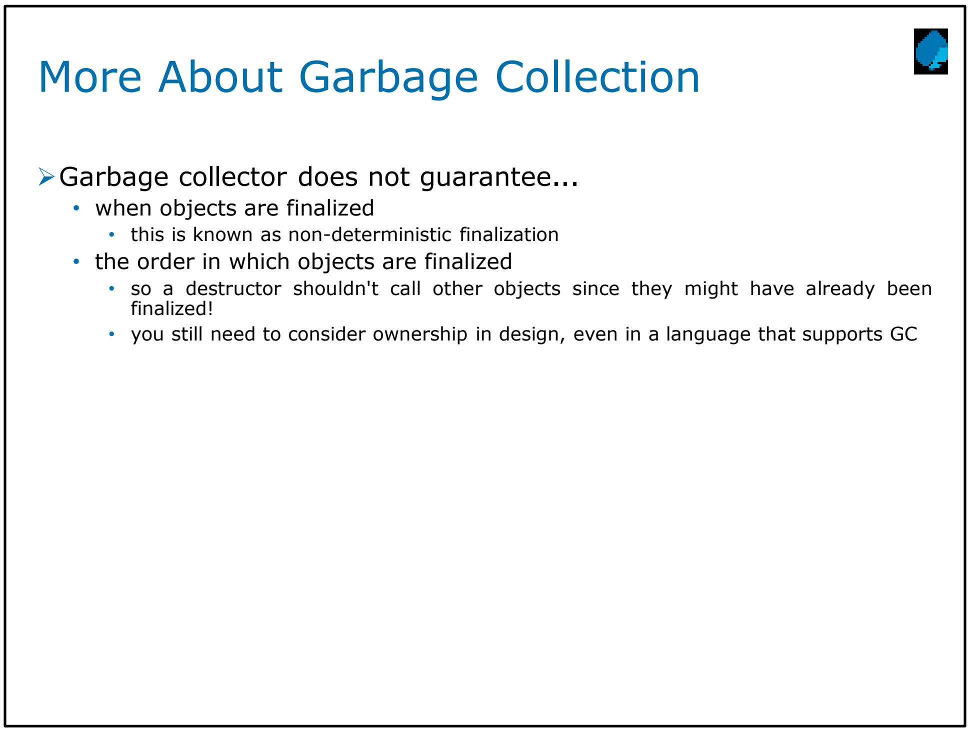
Finalize:

Finalize() belongs to the Object class.
It is automatically called by the Garbage Collection mechanism when the object goes out of the scope(usually at the end of the program
It is slower method and not suitable for instant disposing of the objects.
It is non-deterministic function i.e., it is uncertain when Garbage Collector will call Finalize() method to reclaim memory.

Dispose:

Dispose() belongs to the IDisposable interface
We have to manually write the code to implement it.
E.g.- If we have Employee class we need to implement the IDisposable interface and write code. We may have to suppress the Finalize method
Using GC.SuppressFinalize() method.
Faster method for instant disposal of the objects.

It is deterministic function as Dispose() method is explicitly called by the User Code.



More About Garbage Collection

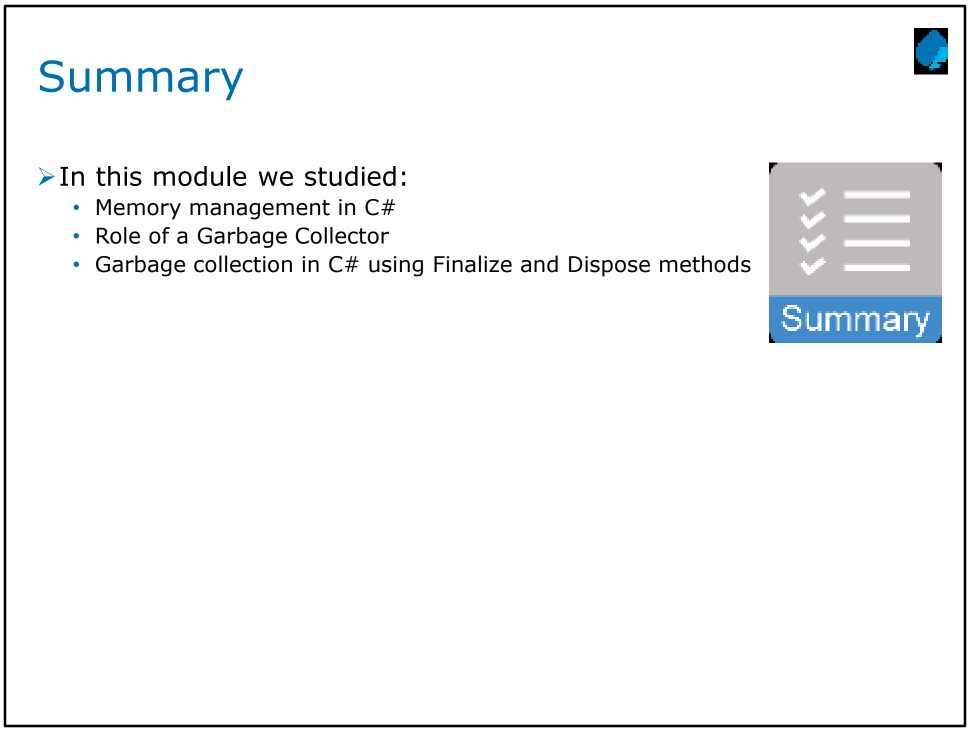
- Garbage collector does not guarantee...
  - when objects are finalized
    - this is known as non-deterministic finalization
  - the order in which objects are finalized
    - so a destructor shouldn't call other objects since they might have already been finalized!
    - you still need to consider ownership in design, even in a language that supports GC

More About Garbage Collection:

C# does not follow the C++ model of destruction. In C# destruction is non-deterministic. This means that while it is possible to write code that will definitely be called when the object is destroyed (via a destructor) you cannot control exactly when the object will be destroyed. The C# garbage collector does not destroy an object the moment that the object becomes unreachable.
This is different to C++. In C++ finalization is deterministic. That is, in C++ the exact moment when an object is destroyed is known precisely (but the price paid for this guarantee is that some destructions have to be explicitly requested by the programmer using a delete expression).
As well as not guaranteeing when an object will be finalized, C# also does not specify the order in which objects are finalized. Just because one object is created before another object does not mean that the first object is finalized either before or after the second object. The order is unspecified.

Demo

Add the notes here.



Summary

- In this module we studied:
  - Memory management in C#
  - Role of a Garbage Collector
  - Garbage collection in C# using Finalize and Dispose methods

Add the notes here.

Answers for the Review Questions:

1) Memory Management in C# is automated process which is done using garbage collector. When no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object is released. This recycled memory can then be used for a subsequent allocation.

2) SuppressFinalize() method can be called inside Dispose() method to prevent the garbage collector from running when Dispose() method is called.

Review Question

- How does memory management take place in C#?
- What is the use of SuppressFinalize method of GC class?

Add the notes here.