CSc 346
Object Oriented Programming

Ken Gamradt

Spring 2024

Chapter 6 – Part 2

Managing memory with reference and value types

- There are two categories of memory:
  - **stack** memory
  - **heap** memory
- With modern operating systems, the stack and heap can be anywhere in physical or virtual memory
- Stack memory is faster to work with
  - it is managed directly by the CPU and because it uses a last-in, first-out mechanism
  - it is more likely to have the data in its L1 or L2 cache
  - limited in size
- Heap memory is slower to work with
  - it is much more plentiful
- Using a macOS terminal, enter the command **ulimit -a** to discover that the stack size is limited to 8,192 KB and that other memory is "unlimited”
- This limited amount of stack memory is why it is so easy to fill it up and get a "stack overflow"

Defining reference and value types

- There are three C# keywords that you can use to define object types:
  - class
  - record
  - struct
- All can have the same members, such as fields and methods
- One difference between them is how memory is allocated
- When you define a type using **record** or **class** – defining a **reference type**
  - This means that the memory for the object itself is allocated on the **heap**
    - memory address of the object (a little overhead) is stored on the stack
- When you define a type using **record struct** or **struct** – defining a **value type**
  - This means that the memory for the object itself is allocated on the **stack**

Defining reference and value types

- If a struct uses field types that are not of the struct type, then those fields will be stored on the heap
    - the data for that object is stored in both the stack and the heap!
- These are the most common struct types:
    - **Number** System **types**: byte, sbyte, short, ushort, int, uint, long, ulong, float, double, and decimal
    - **Other** System **types**: char, DateTime, and bool
    - System.Drawing **types**: Color, Point, and Rectangle
- Almost all the other types are class types, including string
- The other major difference is that you cannot inherit from a struct

How reference and value types are stored in memory

- The number1 variable is a **value type** (also known as struct) so it is allocated on the stack, and it uses 4 bytes of memory since it is a 32-bit integer
- Its value, 49, is stored directly in the variable

```
int number1 = 49;
```

- The number2 variable is also a **value type** so it is also allocated on the stack, and it uses 8 bytes since it is a 64-bit integer

```
long number2 = 12;
```

- The location variable is also a **value type** so it is allocated on the stack, and it uses 8 bytes since it is made up of two 32-bit integers, x and y

```
System.Drawing.Point location = new(x: 4, y: 5);
```

How reference and value types are stored in memory

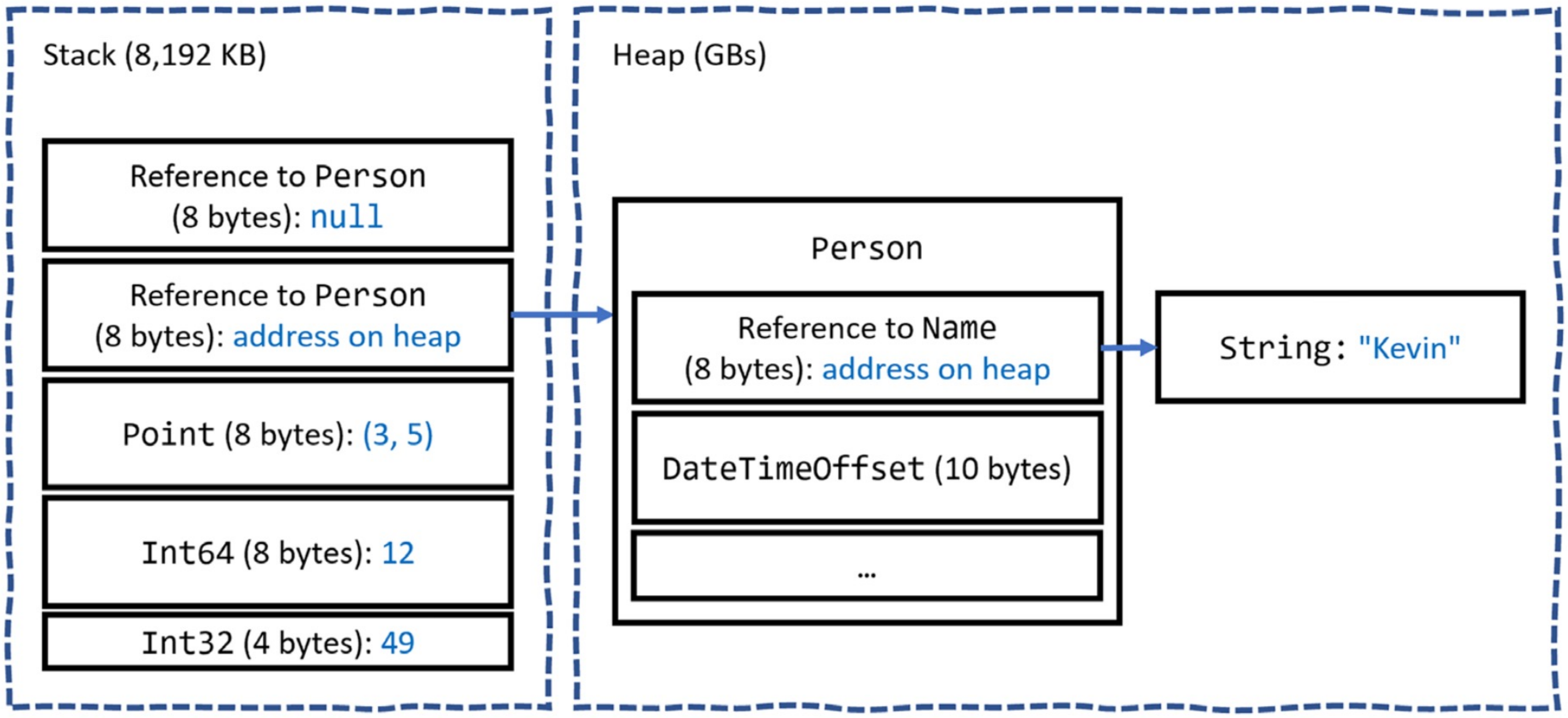
- The kevin variable is a **reference type** (also known as class) so 8 bytes for a 64-bit memory address (assuming a 64-bit operating system) is allocated on the stack and enough bytes on the heap to store an instance of a Person

```
Person kevin = new()
    { Name = "Kevin", DateOfBirth = new(year: 1988, month: 9, day: 23) };
```

- The sally variable is a **reference type** so 8 bytes for a 64-bit memory address is allocated on the stack
- It is currently **null**, no memory has yet been allocated for it on the heap

```
Person sally;
```

How reference and value types are stored in memory


Stack (8,192 KB)
Reference to Person (8 bytes): null
Reference to Person (8 bytes): address on heap
Point (8 bytes): (3, 5)
Int64 (8 bytes): 12
Int32 (4 bytes): 49
Heap (GBs)
Person
Reference to Name (8 bytes): address on heap
DateTimeOffset (10 bytes)
...
String: "Kevin"

How reference and value types are stored in memory

- All the allocated memory for a reference type is stored on the heap
- If a value type such as DateTimeOffset is used for a field of a reference type like Person, then the DateTimeOffset value is stored on the heap
- If a value type has a field that is a reference type, then that part of the value type is stored on the heap
- Point is a value type that consists of two fields, both of which are themselves value types, so the entire object can be allocated on the stack
- If the Point value type had a field that was a reference type, like string, then the string bytes would be stored on the heap

How reference and value types are stored in memory

- When the method completes, all the stack memory is automatically released from the top of the stack
- However, heap memory could still be allocated after a method returns
- It is the .NET runtime garbage collector's responsibility to release this memory at a future date
- Heap memory is not immediately released to improve performance
- Stack memory is literally a stack: memory is allocated at the top of the stack and removed from there when it is no longer needed
- C# developers do not have control over the allocation or release of memory
- Memory is automatically allocated when methods are called, and that memory is automatically released when the method returns
- This is known as **verifiably safe code**

Understanding boxing

- Boxing in C# is when a value type is moved to heap memory and wrapped inside a System.Object instance
- Unboxing is when that value is moved back onto the stack
- Unboxing happens explicitly
- Boxing happens implicitly, so it can happen without the developer realizing
- Boxing can take up to 20 times longer than without boxing

Understanding boxing

- An int value can be boxed and then unboxed

```
int n = 3;
object o = n;     // Boxing happens implicitly
n = (int)o;       // Unboxing only happens explicitly
```

- A common scenario is passing value types to formatted strings

```
// not boxed because string is a reference type and therefore already on the heap
string name = "Hilda";
DateTime hired = new(2024, 2, 21); int days = 5;
// hired and days are value types that will be boxed
Console.WriteLine("{0} hired on {1} for {2} days.", name, hired, days);
```

Understanding boxing

- Boxing and unboxing operations have a negative impact on performance
- Although it can be useful for a .NET developer to be aware of and to avoid boxing, for most .NET project types and for many scenarios, boxing is not worth worrying too much about because the overhead is dwarfed by other factors like making a network call or updating the user interface

Equality of types

- It is common to compare two variables using the == and != operators
- The behavior of these operators is different for reference types and value types

- When you check the equality of two value type variables, .NET literally compares the values of those two variables on the stack and returns true if they are equal

```
int a = 3;
int b = 3;
WriteLine($"a == b: {(a == b)}");   // true
```

Equality of types

- It is common to compare two variables using the == and != operators
- The behavior of these operators is different for reference types and value types

- When you check the equality of two reference type variables, .NET compares the memory addresses of those two variables and returns true if they are equal

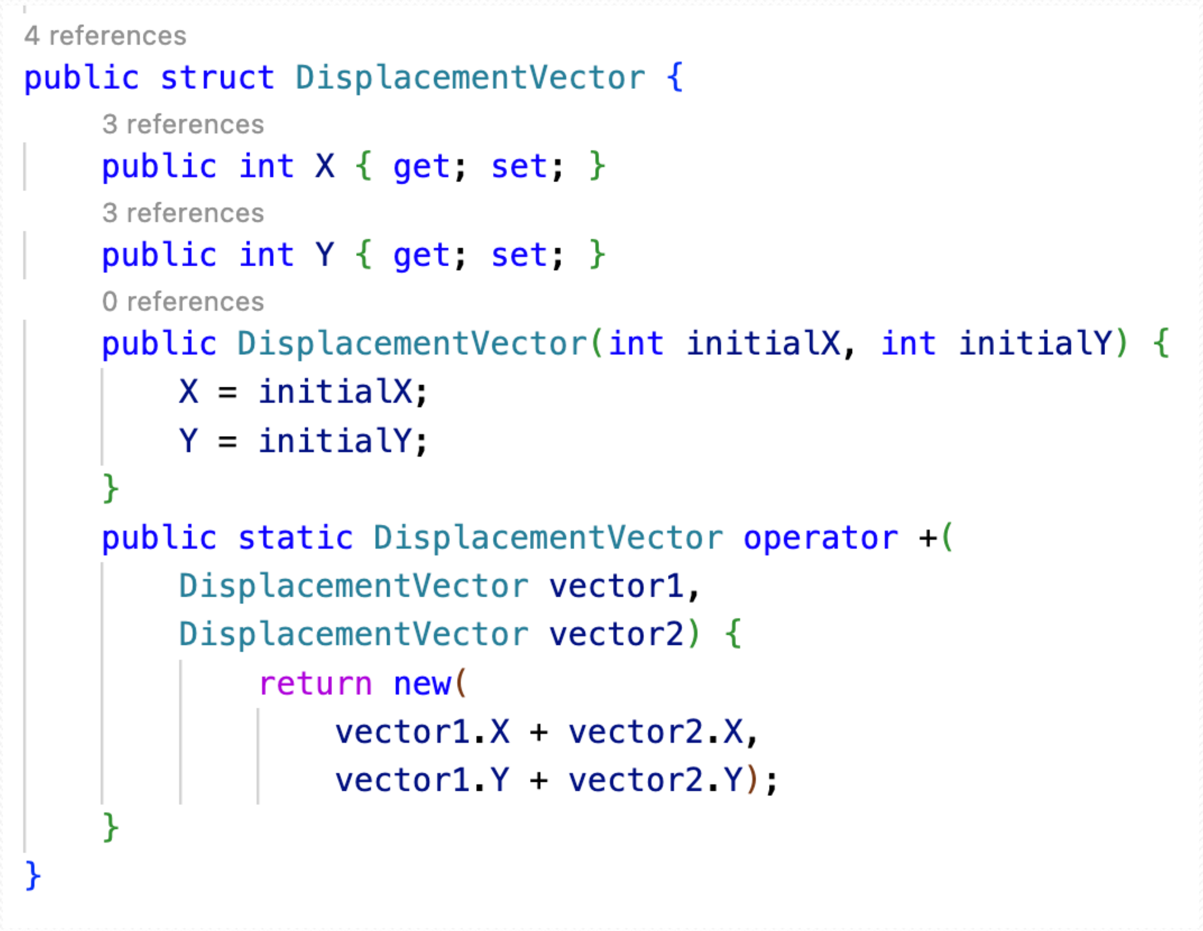
```
Person p1 = new() { Name = "Kevin" };
Person p2 = new() { Name = "Kevin" };
WriteLine($"p1 == p2: {(p1 == p2)}");          // false
```

- This is because they are not the same object
  - They have different memory addresses

Defining struct types

- The type is declared using struct instead of class
- It has two int properties, named X and Y, that will auto-generate two private fields with the same data type, which will be allocated on the stack
- It has a constructor to set initial values for X and Y
- It has an operator to add two instances together that returns a new instance of the type, with X added to X, and Y added to Y

Defining struct types



```
4 references
public struct DisplacementVector {
    3 references
    public int X { get; set; }
    3 references
    public int Y { get; set; }
    0 references
    public DisplacementVector(int initialX, int initialY) {
        X = initialX;
        Y = initialY;
    }
    public static DisplacementVector operator +(
        DisplacementVector vector1,
        DisplacementVector vector2) {
            return new(
                vector1.X + vector2.X,
                vector1.Y + vector2.Y);
    }
}
```

Defining record struct types

- C# 10 introduced the ability to use the **record** keyword with **struct** types as well as with **class** types
- We could define the DisplacementVector type

```
public record struct DisplacementVector(int X, int Y);
```

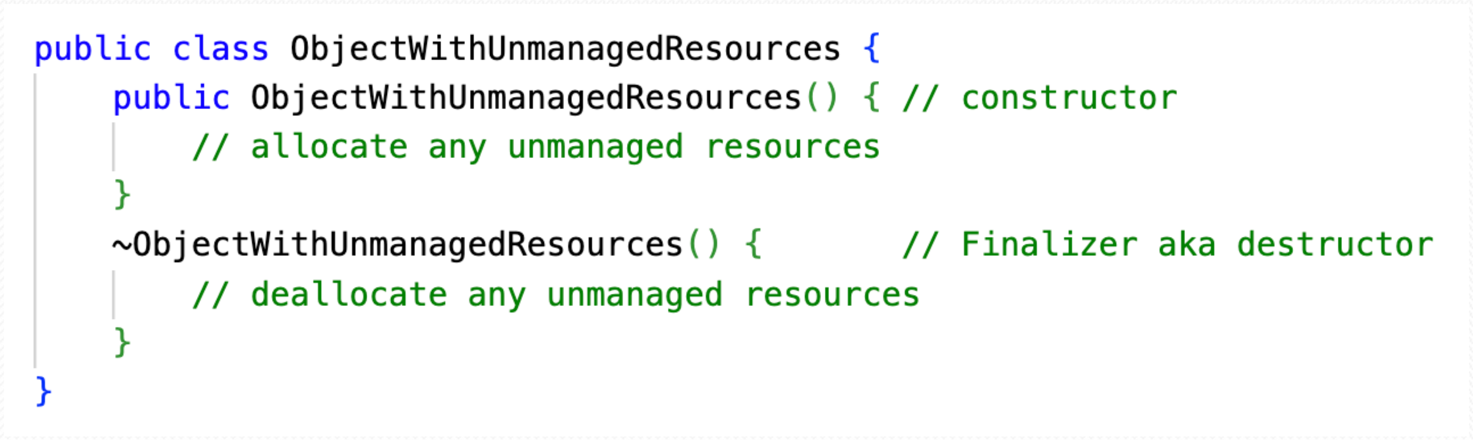
- With this change, Microsoft recommends explicitly specifying class if you want to define a record class even though the class keyword is optional

```
public record class ImmutableAnimal(string Name);
```

Releasing unmanaged resources

- We've seen that constructors can be used to initialize fields and that a type may have multiple constructors
- Imagine that a constructor allocates an **unmanaged resource**
  - Anything that is not controlled by .NET, such as a file or mutex under the control of the operating system
- Unmanaged resources **must be manually released** because .NET cannot do it for us using its automatic garbage collection feature   // C++ is unmanaged
- Garbage collection is an advanced topic
  - We will look at some code examples
  - You do not need to write the code yourself
- Each type can have a single **finalizer** (**destructor**) that will be called by the .NET runtime when the resources need to be released
- A finalizer has the same name as a constructor
  - Type name, but it is prefixed with a tilde, ~

Releasing unmanaged resources



```
public class ObjectWithUnmanagedResources {
    public ObjectWithUnmanagedResources() { // constructor
        // allocate any unmanaged resources
    }
    ~ObjectWithUnmanagedResources() {       // Finalizer aka destructor
        // deallocate any unmanaged resources
    }
}
```

- This is the minimum you should do when working with unmanaged resources
- The problem with only providing a finalizer is that the .NET garbage collector requires two garbage collections to completely release the allocated resources
- It is recommended to also provide a method to allow a developer who uses your type to explicitly release resources so that the garbage collector can release managed parts of an unmanaged resource

Releasing unmanaged resources

```
public class ObjectWithUnmanagedResources : IDisposable {
    public ObjectWithUnmanagedResources() {
        // allocate unmanaged resource
    }
    ~ObjectWithUnmanagedResources() { // Finalizer
        Dispose(false);
    }
    bool disposed = false; // have resources been released?
    public void Dispose() {
        Dispose(true);
        // tell garbage collector it does not need to call the finalizer
        GC.SuppressFinalize(this);
    }
    protected virtual void Dispose(bool disposing) {
        if (disposed) return;
        // deallocate the *unmanaged* resource
        // ...
        if (disposing) {
            // deallocate any other *managed* resources
            // ...
        }
        disposed = true;
    }
}
```

Releasing unmanaged resources

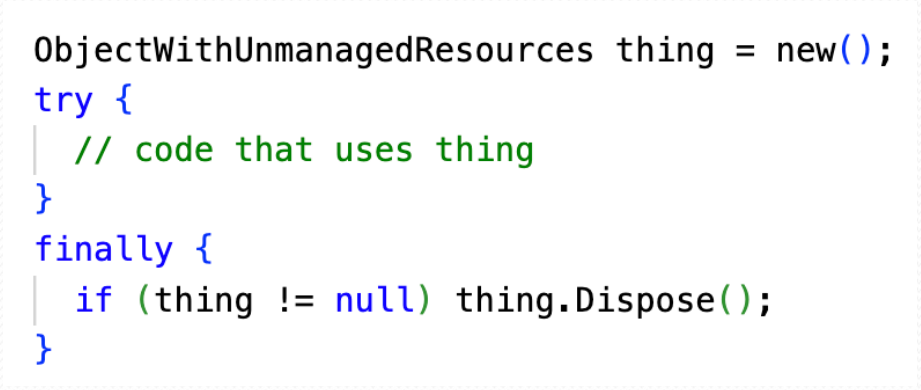
- There are two Dispose methods, one public and one protected:
  - public void Dispose method will be called by a developer using your type
    - Here, both unmanaged and managed resources need to be deallocated
  - The protected virtual void Dispose method with a bool parameter is used internally to implement the deallocation of resources
    - It needs to check the disposing parameter and disposed field because if the finalizer thread has already run and it called the ~Animal method, then only unmanaged resources need to be deallocated
- The call to GC.SuppressFinalize(this) is what notifies the garbage collector that it no longer needs to run the finalizer
  - Removes the need for a second garbage collection

Ensuring that Dispose is called

- When someone uses a type that implements IDisposable, they can ensure that the public Dispose method is called with the using statement

```
using (ObjectWithUnmanagedResources thing = new()) {// code that uses thing}
```

- The compiler converts your code into something like the following, which guarantees that even if an exception occurs, the Dispose method will still be called



```
ObjectWithUnmanagedResources thing = new();
try {
  // code that uses thing
}
finally {
  if (thing != null) thing.Dispose();
}
```

Working with null values

- What if a variable does not yet have a value?
- How can we indicate that?
- C# has the concept of a null value, which can be used to indicate that a variable has not been set

Making a value type nullable

- By default, **value types** like int and DateTime must always have a value
- Sometimes, for example, when reading values stored in a database that allows empty, missing, or null values, it is convenient to allow a value type to be null
- We call this a **nullable value type**
- You can enable this by adding a **question mark** as a suffix to the type when declaring a variable

Making a value type nullable

```
int thisCannotBeNull = 4;
thisCannotBeNull = null;                              // compile error!
WriteLine(thisCannotBeNull);

int? thisCouldBeNull = null;
WriteLine(thisCouldBeNull);                           // null value
WriteLine(thisCouldBeNull.GetValueOrDefault());       // 0

thisCouldBeNull = 7;
WriteLine(thisCouldBeNull);                           // 7
WriteLine(thisCouldBeNull.GetValueOrDefault());       // 7
```

Understanding null-related initialisms

Initialism	**Meaning**	**Description**
NRT	Nullable reference types	A compiler feature introduced with C# 8 and enabled by default in new projects with C# 10 that performs static analysis of your code at design time and shows warnings of potential misuse of `null` values for reference types.
NRE	`NullReferenceException`	An exception thrown at runtime when **dereferencing** a `null` value, aka accessing a variable or member that has a `null` value.
ANE	`ArgumentNullException`	An exception thrown at runtime by a method invocation when an argument has a `null` value when that is not valid.

Understanding nullable reference types

- The use of the null value is so common, in so many languages, that many experienced programmers never question the need for its existence
- There are many scenarios where we could write better, simpler code if a variable is not allowed to have a null value
- The most significant change to the language in C# 8.0 was the introduction of nullable and non-nullable reference types
- You are probably thinking that "Reference types are already nullable!"
- And you would be right, but in C# 8.0 and later, reference types can be configured to no longer allow the null value by setting a file-level or project-level option to enable this useful new feature
- Since this is a big change for C#, Microsoft decided to make the feature opt-in

Understanding nullable reference types

- It will take years for this new C# language feature to make an impact since thousands of existing library packages and apps will expect the old behavior
- Even Microsoft did not have time to fully implement this new feature in all the main .NET packages until .NET 6
- Important libraries like Microsoft.Extensions for logging, dependency injections, and configuration were not annotated until .NET 7
- You can choose between several approaches for your own projects
  - **Default**: No changes are needed
    - Non-nullable reference types are not supported
  - **Opt-in project, opt-out files**: Enable the feature at the project level and, for any files that need to remain compatible with old behavior, opt out
    - Approach Microsoft is using internally while it updates its own packages
  - **Opt-in files**: Only enable the feature for individual files

Controlling the nullability check feature

- To enable the feature at the project level, add the following to your project file:

```
<PropertyGroup>
  ...
  <Nullable>enable</Nullable>
</PropertyGroup>
```

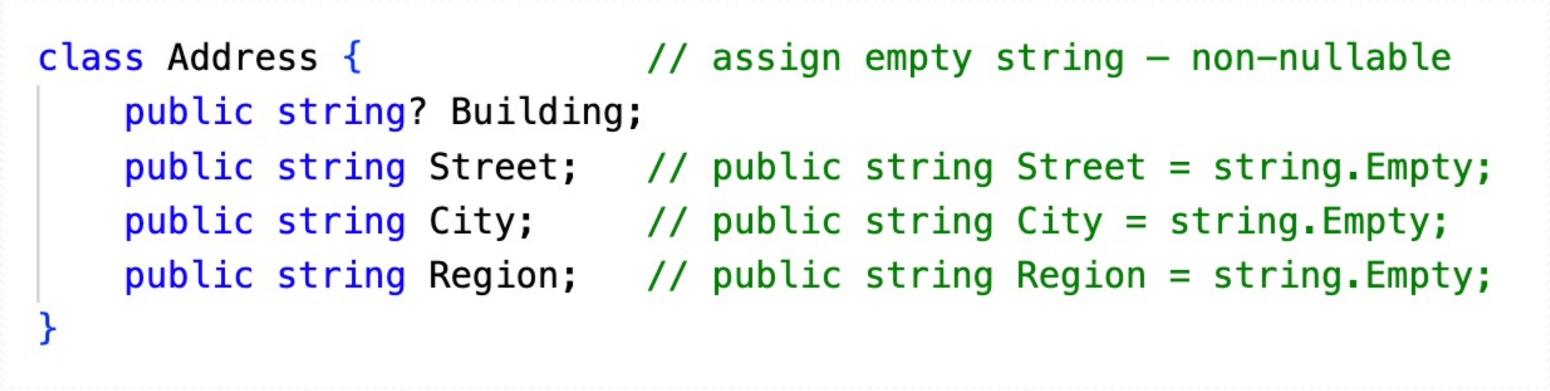
```
<PropertyGroup>
    ...
    <Nullable>disable</Nullable>
</PropertyGroup>
```

- This is now done by default in project templates that target .NET 6.0.

- To disable the feature at the file level, add the following to the top of a code file:
  #nullable disable
- To enable the feature at the file level, add the following to the top of a code file:
  #nullable enable

Declaring non-nullable variables and parameters

- If you enable nullable reference types and you want a reference type to be assigned the null value, then you will have to use the same syntax as making a value type nullable, that is, adding a ? symbol after the type declaration
- How do nullable reference types work?
- Let's look at an example
- When storing information about an address, you might want to force a value for the street, city, and region, but the building can be left blank, that is, null



```
class Address {                 // assign empty string – non-nullable
    public string? Building;
    public string Street;   // public string Street = string.Empty;
    public string City;     // public string City = string.Empty;
    public string Region;   // public string Region = string.Empty;
}
```

Checking for null

- Checking whether a nullable reference type or nullable value type variable currently contains null is important because if you do not, a NullReferenceException can be thrown, which results in an error

```
if (thisCouldBeNull != null) {              // access a member of thisCouldBeNull
    int length = thisCouldBeNull.Length;         // could throw exception
}
```

- C# 7.0 introduced is combined with the ! (not) operator as an alternative to !=

```
 if ( ! (thisCouldBeNull is null)) { …
```

- C# 9.0 introduced is not as an even clearer alternative

```
if (thisCouldBeNull is not null) { …
```

Checking for null

- If you are trying to use a member of a variable that might be null

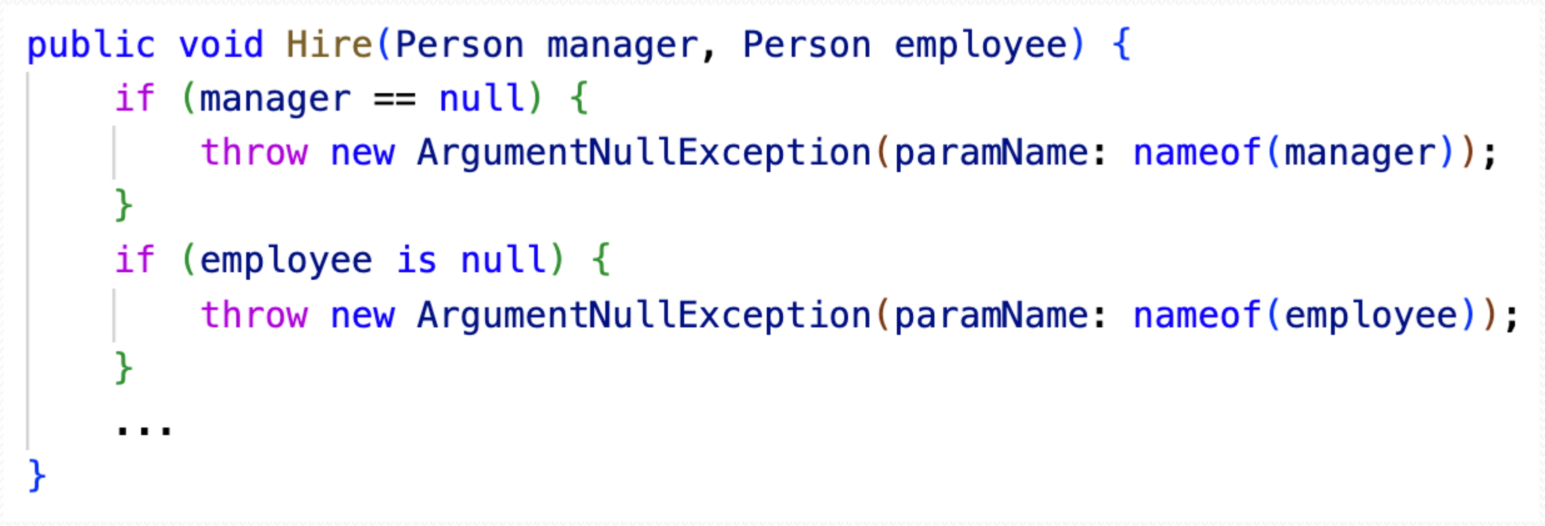
```
string authorName = null;
int? authorNameLength;
// The following throws a NullReferenceException
authorNameLength = authorName.Length;
// Instead of throwing an exception, null is assigned
authorNameLength = authorName?.Length; // null-conditional operator, ?.
```

- Sometimes you want to either assign a variable to a result or use an alternative value, such as 3, if the variable is null

```
// Result will be 25 if authorName?.Length is null
authorNameLength = authorName?.Length ?? 25; // null-coalescing operator, ??
```

Checking for null in method parameters

- When defining methods with parameters, it is good to check for null values
- In earlier versions of C#, you would write if statements to check for null parameter values and then throw an ArgumentNullException



```
public void Hire(Person manager, Person employee) {
    if (manager == null) {
        throw new ArgumentNullException(paramName: nameof(manager));
    }
    if (employee is null) {
        throw new ArgumentNullException(paramName: nameof(employee));
    }
    ...
}
```

Checking for null in method parameters

- C# 10 introduced a method to throw an exception if an argument is null

```
public void Hire(Person manager, Person employee) {
    ArgumentNullException.ThrowIfNull(manager);
    ArgumentNullException.ThrowIfNull(employee);
    ...
}
```

- C# 11 previews introduced a new !! operator that does this for you

```
public void Hire(Person manager!!, Person employee!!) {
    ...
}
```

Checking for null in method parameters

- The if statement and throwing of the exception are done for you
- The code is injected and executes before any statements that you write
- This syntax is controversial within the C# developer community
- Some would prefer the use of attributes to decorate parameters instead of pairs of characters
- The .NET product team claims to have saved more than 10,000 lines of code throughout the .NET libraries by using this feature
- That sounds like a good reason to use it to me!
- And no one must use it if they choose not to

- Unfortunately, the team eventually decided to remove the feature, so now we all have to write the null checks manually
- https://devblogs.microsoft.com/dotnet/csharp-11-preview-updates/#removeparameter-null- checking-from-c-11