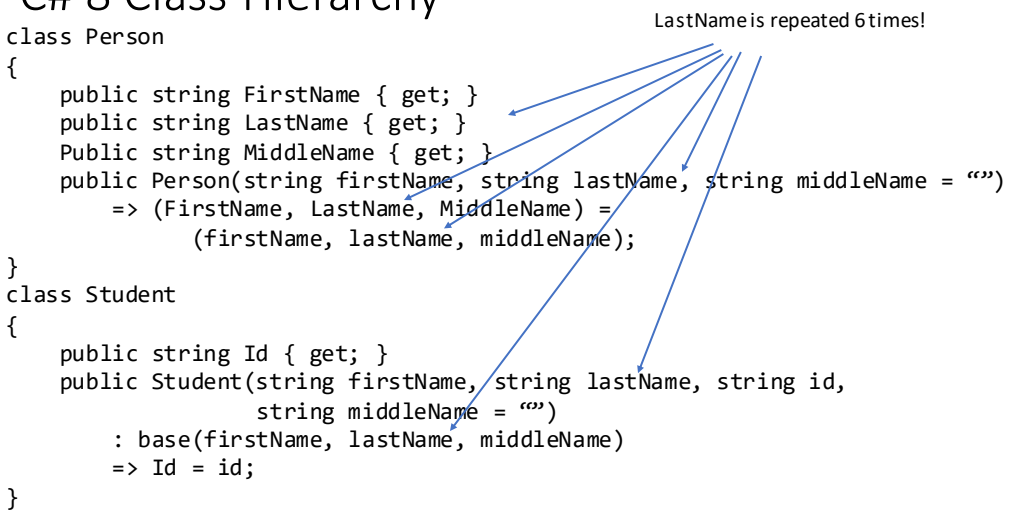
Required Properties

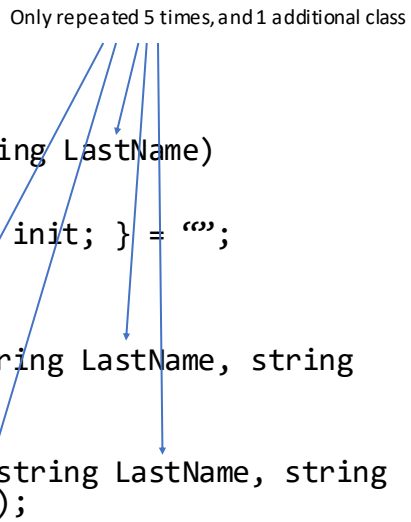
(Or: Initialization Debt)

C# 8 Class Hierarchy



In the base type, we have to state LastName 4 times!
2 times in every derived type
MiddleName has to be constantly restated, even though it's optional

C# 9 Record Hierarchy




```
record Person(string FirstName, string LastName)
{
    public string MiddleName { get; init; } = “”;
}

record Student(string FirstName, string LastName, string
Id) : Person(FirstName, LastName);

record Professor(string FirstName, string LastName, string
Title) : Person(FirstName, LastName);
```

We got rid of restating the optional MiddleName, and removed the 3 restatements of LastName from the base class. However, the subclasses still require LastName to be restated twice.

C# 9 Nominal Record Hierarchy

```
record Person
{
    public string FirstName { get; init; }
    public string MiddleName { get; init; } = “”;
    public string LastName { get; init; }
}
record Student: Person
{
    public string Id { get; init; }
}
record Professor : Person
{
    public string Title { get; init; }
}
```

We don’t need to restate LastName at every level: good! However, we’ve lost an important requirement, as LastName is no longer required to be set at the creation site, and we get nullable warnings on the classes

What is a constructor?

- A constructor is the set of information required to construct a class
  - Parameter Types
  - Parameter Names
  - Ordering
- Or: a *contract* for construction

```
public Student(
    string firstName,
    string lastName,
    string middleName = “”)
```

Contract: given these inputs, I can construct an output of this type
Ordering makes this positional
Please remember contract. I'm going to use it for the rest of the presentation

Positional Constructors

Pro

- Ability to specify both required and optional parameters
- Nullability of parameters is not conflated with requiredness
- If a new required element is added and constructing code is not updated, we see a runtime failure
- Ability to have multiple contracts

Con

- Order matters: reordering is a breaking change
- Constructors are cumulative. All previous contracts must be restated, even if they are unmodified
- Breaks occur in the derived constructor, not at the point that actually depends on the contract

Conflating nullability with requiredness is a common theme among users
Multiple contracts: think of copy constructors, for example

Glimmers of a solution: Properties

- Concerns around order lead us to examine properties and object initializers as an initialization mechanism
- Pros:
  - No order
  - No restating base class properties required – including when new properties are added!
- Cons:
  - No *contract* – This is the big one, and what we're trying to solve here
    - Let alone multiple contracts

Components of a contract

- Calling a constructor is a form of validating a contract
  - All parameters must be present. If a new parameter has been added or an old parameter has been removed, the runtime will fail
  - All parameters must be the correct type. If a parameter has changed to an incompatible type, the runtime will fail
  - Everywhere that depends on a contract will validate this, directly or indirectly
- However, there is no way of stating "I have whatever contract that thing had, plus my own"
  - All previous terms must be restated.
  - The contract failure doesn't occur at the best place: the component that actually depends on the contract.

Next slide for an example

Restating contracts

```
class Person
{
    public string FirstName { get; }
    public string LastName { get; }
    Public string MiddleName { get; }
    public Person(string firstName, string lastName, string middleName = “”)
        => (FirstName, LastName, MiddleName) =
              (firstName, lastName, middleName);
}
class Student
{
    public string Id { get; }
    public Student(string firstName, string lastName, string id,
                   string middleName = “”)
        : base(firstName, lastName, middleName)
        => Id = id;
}
```

If these are in separate assemblies, and Person is updated, then the contract failure occurs in the base call, despite the fact that Student doesn’t care about the base contract

Multiple Contracts

```
public class Person
{
    public string FirstName { get; required init; }
    public string? MiddleName { get; init; }
    public string LastName { get; required init; }

    // Requires FirstName/LastName to be set by consumers
    public Person() {}

    // Requires nothing to be set by consumers
    public Person(Person other)
    {
        FirstName = other.FirstName;
        MiddleName = other.MiddleName;
        LastName = other.LastName;
    }
}
```

We also need to consider how to ensure that multiple contracts can be created in this property-based world. It's not enough to say "Just these properties are required" Consider also future expansions into factories. We'd want MakeAMads to be able to say "You must specify LastName", but allow it to elide FirstName

Break it up: Nominal contracts

- Take the contracts we have today and break them up.
- Every constructor has a property contract
- That contract states “You must initialize these properties”
- Derived constructors can say “You must initialize my properties, and see BaseContract_1 for the other things you must do”
- Contracts must be validated where they are depended upon
  - If your constructor is only adding to a base contract, *it does not depend on it!*

This means that, if I was creating a new instance of student, I should have to validate that the contract for student and the contract for person are both satisfied *at the construction site*

If I create a new constructor that provides defaults for some base parameters, *then* that constructor would have to validate the contract and export an entirely new contract, because it depends on the contract

Nominal Contracts Implementations

- This is where input is needed
- There are 2 general approaches:
  - Static verification. These approaches force the CLR to call a method, or construct a generic type, that will fail if a contract has changed
  - Manual verification. These approaches track what was been assigned and what has not, then verify that everything that should be set has been set

The first 3 implementation methods we'll go over are static verification

Implementation 1: Contract Types

- Every contract would generate a type, and the constructor would be modreq'd with that type
- That type would contain information in it describing the required properties and types
- Any changes to the contract break the call to the constructor

```
public modreq(typeof(
Requirements_System.String_FirstName_System.String_LastName
)) Person() {}
```

This unfortunately leaves us where we are today: the contract is typed in the constructor, so the base call to a required parameter constructor is a silent restatement of the requirements. If you add a required property to assembly A, assembly B would be required to update before it's usable in assembly C
Type names have a limited size, so this would almost certainly run into issues where we'd have to do things like generating multiple types

Implementation 2: Verification Methods

- In this approach, we'd have a 5 step approach
  1. modreq the constructor with some type that means "You must check a contract"
  2. Produce a method that encodes in its parameters the types and names of the contract elements
  3. Attribute the constructor as requiring that contract method to be called
  4. If any base constructors are called, attribute the constructor as calling those base constructors
  5. At construction site, the compiler emits a call to these methods. If things change, this call will fail at the relevant construction point.

```
public void $Verify(string FirstName, string LastName) {}
```

Benefits: adding a new required property will change the method, creating a missingmethodexception at runtime at the correct location
Making a property optional later could be a breaking change, unless we have a "so, include this in the contract for backcompat but not really enforce it" mechanism

Implementation 3: Variance

```
interface II<in TMember, out TRest> { } // I Initialize
interface IR<TMember, TRest> : // I Require
    II<TMember, TRest>,
    II<object, IR<TMember, TRest>> { }
interface IE : II<object, IE> { } // I Empty
```

- Create a variance scenario where if the requirements change, substitution would no longer work
- Constructor would create a generic method where T is constrained to:
  - `IR<FirstName, IR<LastName, IE>>`
- Call point would then call the method with a type that contains what it initializes:
  - `II<FirstName, II<MiddleName, II<LastName, IE>>>`

If the contract changes, then II will no longer be a valid substitute for IR, and you'll get a type mismatch exception

Pros: could tolerate some property moves to base classes

Cons: actually forcing the runtime to perform this type of type check is hard. Would have to design how it would work with multi-level constructors, and ensure there are no ordering issues.

Implementation 4: Track assignments in the type

- Create a post-constructor method that must be called by the constructing point
- The type will track backing field initializations (maybe a bit array)
- Post validation will be responsible for throwing

We liked the idea of validators before, this would be a need to readd them.
Resilient to most hierarchy changes: moving a property to a base type could be done safely, and making it optional
Could be expensive for structs: an additional storage requirement might mean that most structs and the BCL are unable to adopt this

Implementation 5: Track assignments outside the type

- Each type would create a class or struct that tracks assignments
- When creating an instance of a type, we silently create one of these tracking types, and track the assignments in it.
- After construction, we call a method on the tracking type to verify that all requirements were met

Pros: No size bloat in classes and structs
Resilient to moving properties between class levels
Cons: Very leaky abstraction. We'll have to thread this through potentially many constructor levels, future factory work, type classes…
This basically brings back the record builder approach.

Implementation 6: Magic Cookies

- We create a stable algorithm to generate a magic cookie value from the requirements
- This cookie is passed as part of a constructor
- If the cookie does not validate, we throw

This is a very simple, but very brittle approach. I'm including it here as an extreme end, but I don't actually expect us to choose it

Implementation N/A: No verification

- To put all the options on the table, we could have no verification of the contract at all
- No runtime failures when contracts are violated
- Version updates might introduce source-breaking changes that do not translate into runtime-breaking changes

Putting it all together: Initialization Debt

1. The user declares that some properties must be set. This is analogous to a user saying that some parameters are optional
   - This is the debt
2. Every constructor exports some contract, allowing it to say "Users must set these things"
   - Letting users know about the debt they'll incur if they use this constructor
3. We have some kind of verification method
   - Ensuring that the debt is paid