Module 2

Derived records & inheritance

Derived records

Like it was mentioned in the previous module, a **record** type supports inheritance.

```
public record Person(string FirstName, string LastName);
public record Teacher(int YearsOfExperience, string FirstName, string LastName) : Person(FirstName, LastName);

public static void Main()
{
  var teacher = new Teacher(10, "Lorem", "Ipsum");
  Console.WriteLine(teacher); // Teacher { FirstName = Lorem, LastName = Ipsum, YearsOfExperience = 10 }
}
```

Derived records

The **record** type can also be **abstract** or **sealed** and may have **abstract** or **virtual** members, like methods

```
public abstract record Person(string FirstName, string LastName)
{
    public abstract int GetSalary();
}

public sealed record Teacher(int YearsOfExperience, string FirstName, string LastName) : Person(FirstName, LastName)
{
    public override int GetSalary() => YearsOfExperience * 2000;
}

public static void Main()
{
  var teacher = new Teacher(10, "Lorem", "Ipsum");
  Console.WriteLine(teacher.GetSalary()); // 20000
}
```

Derived records

A **record** from another **record**. A **record** from a **class**:

```
public class Person
{
  public Person(string firstName, string lastName)
  {
  }
}

public record Teacher(int YearsOfExperience, string FirstName, string LastName) : Person(FirstName, LastName);
// Error: Records may only inherit from object or another record
```

and a **class** from a **record**:

```
public record Person(string FirstName, string LastName);

public class Teacher : Person
{
  // Error: Only records may inherit from records
  public Teacher(int yearsOfExperience, string firstName, string lastName)
  {
    [...]
  }
}
```

Equality

> *"For two record variables to be equal, the run-time type must be equal. The types of the containing variables might be different"* - MSDN

```
public abstract record Person(string FirstName, string LastName);

public sealed record Teacher(string FirstName, string LastName) : Person(FirstName, LastName);
public sealed record Student(string FirstName, string LastName) : Person(FirstName, LastName);

public static void Main()
{
  Person teacher = new Teacher("Lorem", "Ipsum");
  Person student1 = new Student("Lorem", "Ipsum");
  Console.WriteLine(teacher == student1); // false

  Student student2 = new Student("Lorem", "Ipsum");
  Console.WriteLine(student1 == student2); // true
}
```

Equality

To implement this functionality, the C# compiler generates a property of type **`EqualityContract`** which returns a **`type`** object of the record. It allows the equality logic to use the runtime type of the record for checking the equality.

If the base type of a record is an object, this property is virtual.

```
public abstract record Person(string FirstName, string LastName);
// compiles into

public abstract class Person : IEquatable<Person>
{
  protected virtual Type EqualityContract
  {
    [CompilerGenerated]
    get
    {
      return typeof(Person);
    }
  }
  [...]
}
```

Equality

If the base type is another record type, this property is an override. When a record type is sealed, the property is sealed.

```
public sealed record Student(string FirstName, string LastName) : Person(FirstName, LastName);
// compiles into

public sealed class Student : Person, IEquatable<Student>
{
  protected override Type EqualityContract
  {
    [CompilerGenerated]
    get
    {
      return typeof(Student);
    }
  }
}
```

with expression

As we learned in the previous module, **with** expression uses the compiler-generated **Clone** method. This method uses a covariant return type, so the result of the **with** expression has the same type as the expression subject.

All runtime properties are copied, however, you can set only the compile-time properties. Let's take a look at the following example:

```
public abstract record Person(string FirstName, string LastName);
public sealed record Student(string FirstName, string LastName, int Year) : Person(FirstName, LastName);

public static void Main()
{
  Person student = new Student("Lorem", "Ipsum", 1);
  Person harryPotter = student with { FirstName = "Harry", LastName = "Potter" };
  Console.WriteLine(harryPotter is Student); // true

  Person copiedStudent = student with { Year = 6 };   // Error: Person does not contain a definition for 'Year'
}
```

The **copiedStudent** has **Student** type, however, you cannot set **Student** type properties like **Year**, because they are known only during the runtime.

ToString()

`ToString()` and **`PrintMembers()`** methods are also applied to the records with hierarchy. The **`PrintMembers`** method of a derived record type calls the base implementation and collects print information about all properties.

All public properties and fields of all derived and base types are included in the **`ToString`** output, like in the following example:

```csharp
public static void Main()
{
    var harryPotter = new Student("Harry", "Potter", 1);
    Console.WriteLine(harryPotter);
    // Student { FirstName = Harry, LastName = Potter, Year = 1 }
}
```

You can customize the behavior of the **`PrintMembers`** in the same way as it was shown in the previous module.

Deconstrucion

The generated **Deconstruct** method returns all positional properties of the record known at the compile-time. If the record variable type is a base class, the **Deconstruct** method returns only properties known for the base class.

If you want to use **Deconstruct** method of the derived type, the object needs a cast to the derived type.

```
public abstract record Person(string FirstName, string LastName);
public sealed record Student(string FirstName, string LastName, int Year) : Person(FirstName, LastName);

public static void Main()
{
    Person harryPotter = new Student("Harry", "Potter", 1);
    var (firstName, lastName) = harryPotter;
    var (firstName1, lastName2, year) = (Student)harryPotter;
}
```