

# Record Types



**Filip Ekberg**

Principal Consultant & CEO

@fekberg | [fekberg.com](http://fekberg.com)

# Records in C#

Record can be used instead of a class

If the purpose is to represent data

Less code!

Records come with generated code that you normally have to write yourself



# Introducing a Record



# Introducing a Record

```
record Customer();
```



# Introducing a Record

```
record Customer();
```

```
record class Customer();
```



# Introducing a Record

```
record Customer();
```

```
record class Customer();
```

```
record struct Customer();
```



# Introducing a Record

```
record Customer();
```



Reference type

```
record class Customer();
```



```
record struct Customer();
```

Value type



# Records can be defined with a primary constructor



# Record with a Primary Constructor

```
record Customer(string Firstname, string Lastname);
```



# Record with a Primary Constructor

```
record Customer(string Firstname, string Lastname);
```



This is a positional syntax that creates  
a positional record



# Record with a Primary Constructor

```
record Customer(string Firstname, string Lastname);  
  
Customer customer = new("Filip", "Ekberg");
```



**Records can have a body  
where you define other  
properties, methods and fields**



# Positional Record

```
record Customer(string Firstname, string Lastname);  
  
Customer customer = new("Filip", "Ekberg");  
  
Console.WriteLine($"{customer.Firstname} {customer.Lastname}");
```



# Primary Constructor

```
record Customer(string Firstname, string Lastname);  
  
Customer customer = new("Filip", "Ekberg");
```



# Primary Constructor

```
record Customer(string Firstname, string Lastname);  
Customer customer = new("Filip", "Ekberg");
```



**How is this any better than  
creating a traditional class?**

**There's lots of code generated!**

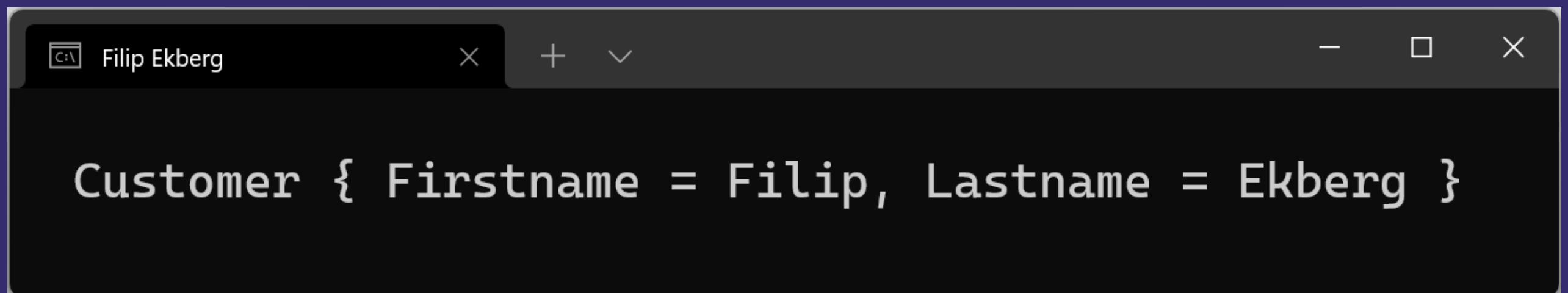


# Records come with value based equality!



```
record Customer(string Firstname, string Lastname);  
  
Customer customer = new("Filip", "Ekberg");  
  
Console.WriteLine(customer);
```

## Record as a String



# Deconstructing a Record

```
record Customer(string Firstname, string Lastname);  
  
Customer customer = new("Filip", "Ekberg");  
  
if(customer is (_, "Ekberg"))  
{  
    ApplyDiscount();  
}
```



# Record with a List

```
record Customer(string Firstname, string Lastname, IList<Order> Orders);  
  
Customer customer = new("Filip", "Ekberg", new List<Order>());
```



# Record with a List

```
record Customer(string Firstname, string Lastname, IList<Order> Orders);  
  
Customer customer = new("Filip", "Ekberg", new List<Order>());  
  
customer.Orders.Add(new());
```



# Record with a List

```
record Customer(string Firstname, string Lastname, IList<Order> Orders);  
  
Customer customer = new("Filip", "Ekberg", new List<Order>());  
  
customer.Orders.Add(new()); ← This will work
```



# Record with a List

```
record Customer(string Firstname, string Lastname, IList<Order> Orders);  
  
Customer customer = new("Filip", "Ekberg", new List<Order>());  
  
customer.Orders.Add(new()); ← This will work  
  
customer.Orders = new List<Order>(); ← This will not work
```



**Records let you focus on the important pieces and skip all the boilerplate code!**



# You cannot use records with Entity Framework



# When to Use Records

When you want to represent data

Not for classes that are meant for business logic



The primary constructor is a  
concise syntax to define the  
least amount of data  
required



If a property uses a record  
you will get value based  
equality for that as well



# Need business logic?

## Use a class instead



This is a good opportunity to validate the refactoring with tests using unit tests



# Where is the generated deconstruct?

If you do not create a primary constructor that will not be generated!



# Record with Optional Parameters

```
public record Order(  
    decimal Total = 0,  
    ShippingProvider ShippingProvider = default,  
    IEnumerable<Item> LineItems = default,  
    bool IsReadyForShipment = true)
```



# Record with Optional Parameters

```
public record Order(  
    decimal Total = 0,  
    ShippingProvider ShippingProvider = default,  
    IEnumerable<Item> LineItems = default,  
    bool IsReadyForShipment = true)  
  
Order order = new()  
{  
    LineItems = items  
};
```



# JSON Property Names

```
{  
  "total": 101,  
  "ready": true,  
  "orderNumber": "24bbea6e-edf7-4657-a925-b875677ef27c"  
}
```



# Record with Attributes

```
public record Order(  
    [property: JsonPropertyName("total")]  
    decimal Total = 0,  
  
    [property: JsonIgnore]  
    ShippingProvider ShippingProvider = default,  
  
    [property: JsonIgnore]  
    IEnumerable<Item> LineItems = default,  
    bool IsReadyForShipment = true);
```



**Which methods do you  
normally want to change?**

**ToString or PrintMembers!**



# Implementing PrintMembers

```
protected virtual bool PrintMembers(StringBuilder builder)
{
    builder.Append("A custom implementation");

    return true;
}
```



**This can be overriden in a  
record that inherits from it!**



# Different Types of Records

```
record Customer(string firstname, string lastname);
```

```
record struct Customer(string firstname, string lastname);
```



# Different Types of Records

```
record Customer(string firstname, string lastname);
```

```
record struct Customer(string firstname, string lastname);
```

```
readonly record struct Customer(string firstname, string lastname);
```



# Record as a Reference Type

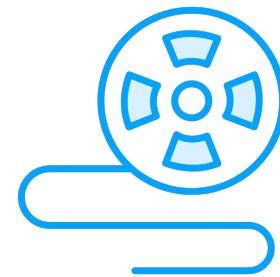
```
record class Customer(string firstname, string lastname);
```



```
record Customer(string firstname, string lastname);
```

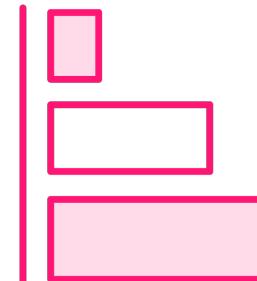


# Creating a Record



**Use a primary constructor**

```
record Customer(string firstname, string lastname)
```



**Also known as a positional record**



**All the properties generate read-only fields  
This an immutable type**



# Record as a String

```
record Customer(string Firstname, string Lastname);  
  
var customer = new Customer("Filip", "Ekberg");  
  
customer.ToString()
```



# Record as a String

```
record Customer(string Firstname, string Lastname);
```

```
var customer = new Customer("Filip", "Ekberg");
```

```
customer.ToString()
```



```
Customer { Firstname = Filip, Lastname = Ekberg }
```



# Value Based Equality

```
record Customer(string Firstname, string Lastname);
```

```
var first = new Customer("Filip", "Ekberg");  
var second = new Customer("Filip", "Ekberg");
```

```
first == second
```



**Values** of each instance is **compared** and this equality comparison returns true



# Record with a Body

```
record Customer(string Firstname, string Lastname)
{
    public string Fullname => $" {Firstname} {Lastname}" ;
} ring()
```



**Records are suitable for  
representing data!**

**Do not use it for business logic**



# Business Logic Should Use Classes

```
class OrderProcessor
{
    public void Process(IEnumerable<Order> orders) { }
}
```



# Combine Classes and Records

```
class OrderProcessor
{
    public void Process(IEnumerable<Order> orders) { }
}

record Order(Guid id, IEnumerable<Item> items);

record Item(Guid id, int quantity, decimal price);
```



# Use **init-only** properties for immutability in classes



# Record Inheritance

```
record Order(Guid id, IEnumerable<Item> items)
```

```
record CancelledOrder(Guid id, IEnumerable<Item> items)  
    : Order(id, items)
```



# Record Inheritance

```
record Order(Guid id, IEnumerable<Item> items)
{
    protected virtual bool PrintMembers(StringBuilder builder) { ... }

record CancelledOrder(Guid id, IEnumerable<Item> items)
    : Order(id, items)
{
    protected override bool PrintMembers(StringBuilder builder)
    {
        return base.PrintMembers(builder);
    }
}
```



# Records Come with Deconstruction

```
record Customer(string Firstname, string Lastname);
```



```
public void Deconstruct(out string Firstname, out string Lastname)
{
    Firstname = this.Firstname;
    Lastname = this.Lastname;
}
```



# Records Come with Deconstruction

```
record Customer(string Firstname, string Lastname);  
  
var customer = new Customer("Filip", "Ekberg");  
  
if(customer is (_, "Ekberg"))  
{ }
```



Add optional parameters  
instead of multiple  
constructors



# Record with Attributes

```
public record Order(  
    [property: JsonPropertyName("total")]  
    decimal Total = 0,  
  
    [property: JsonIgnore]  
    ShippingProvider ShippingProvider = default,  
  
    [property: JsonIgnore]  
    IEnumerable<Item> LineItems = default,  
    bool IsReadyForShipment = true);
```



# Get use to records

You will use them a lot!



**Up Next:**

# **Nullable Reference Types**

---

