Contents

As you start to write your entity creation code, you'll wonder "**where do I put it?**". Where you put it does matter. We strongly recommend that you use a **factory method**. Beyond that, you have choices, each with its pros and cons. We cover some of the choices and considerations in this topic.

Introduction

Where do you put the code that creates an entity? It depends on the entity, the circumstances, and you're preferred style. You have to make your own choices. We'll suggest some ways to think about the choices in this topic.

Whatever you decide to do, we urge you to **create entities within a factory method**.

Use a factory method

A "factory" method is a method that creates an object.

We strongly discourage sprinkling bare creation logic about your code base. You don't want to find a line like this in your application.

```
new Customer(); // more to follow
```

```
New(Customer()  ' more to follow
```

This firm advice applies even if a perfectly well formed Customer is easy to create that way. Sure it is tempting. It is acceptable inside test methods and wherever you create test entities. But don't do this anywhere in the application code base.

Why? Because it never stays this simple. Over time, you and other developers will find multiple, conflicting ways to create a customer that ruin your chances of a consistent, maintainable application. Just say "no".

Here's a simple factory method:

```
public Customer CreateCustomer()
{
 return new Customer(); // more to follow
}
```

```
Public Function CreateCustomer() As Customer
 Return New Customer() ' more to follow
End Function
```

It bears repeating

Do encapsulate creation logic in a factory method

"New" the entity; avoid "CreateEntity(...)"

We recommend that you instantiate the entity with "new" as seen throughout this topic.

DevForce offers an alternative approach using the entity factory build into the EntityManager. The previous example could have been written:

```
public Customer CreateCustomer(EntityManager manager)
{
  return manager.CreateEntity<Customer>(); // more to follow
}
```

```
Public Function CreateCustomer(ByVal manager As EntityManager) As Customer
 Return manager.CreateEntity(Of Customer)() ' more to follow
End Function
```

Although this approach requires the help of an EntityManager, it doesn't actually add the new Customer to the manager. It differs from "new" in these respects:

1. It uses the default constructor internally to instantiate the entity

2. It initializes the entity data properties with the default values defined in the Entity Data Model (if any) but only if the default constructor didn't change them first.
3. It hides a (replaceable) reference to the EntityManager inside the created entity.

Consider a factory without "AddToManager"

The following is advice that stems from a design-friendly and test-oriented mentality.

Remember that adding the new entity to an EntityManager is one of the critical steps to creating an entity. You won't be able to do much with your entity until you take that last step and add the entity to an EntityManager. You won't forget that step if you include it in the factory method:

```
// This is ok
public Customer CreateCustomer(EntityManager manager)
{
  if (null == manager) { /* throw exception */ }
   var cust = new Customer();
  //  other stuff
  manager.AddToManager(cust);
  return cust;
}
```

```
' This is ok
Public Function CreateCustomer(ByVal manager As EntityManager) As Customer
 If Nothing Is manager Then ' throw exception
 End If
Dim cust = New Customer()
 '  other stuff
 manager.AddToManager(cust)
 Return cust
End Function
```

It's ok but it's not a best practice because it combines two distinct concerns: (1) making a valid Customer and (2) integrating with the DevForce infrastructure. That makes entity creation harder to test than it should be. I can't evaluate my Customer creation rules without providing an EntityManager - a requirement that has nothing to do with what I'm trying to test. It also complicates the visual design process in XAML platforms because adding an entity to an EntityManager may be blocked in Blend.

Consider breaking the CreateCustomer method into two overloads:

```
// Can create without adding to manager
public Customer CreateCustomer()
{
   var cust = new Customer();
  // other stuff
  return cust;
}
public Customer CreateCustomer(EntityManager manager)
{
  if (null == manager) { { /* throw exception */ }
   var cust = CreateCustomer();
   manager.AddToManager(cust);
  return cust;
}
```

```
' Can create without adding to manager
Public Function CreateCustomer() As Customer
 Dim cust = New Customer()
 ' other stuff
 Return cust
End Function
Public Function CreateCustomer(ByVal manager As EntityManager) As Customer
 If Nothing Is manager Then ' throw exception
  Dim cust = CreateCustomer()
   manager.AddToManager(cust)
  Return cust
 End If
```

Consider intention-revealing factory methods

In many applications you intentionally create different versions of the same type of entity. You might create "Gold Customers" and "Silver Customers" and "Platinum Customers". You're always creating Customer entities but you configure each variation differently. Perhaps they differ only in a few property values. They could differ more substantially as when creating a "Platinum" customer triggers a sequence of ancillary activities and entity creations that are not available to "Silver" Customers. It might take more information - more and different parameters - to create a "Platinum" customer than a "Silver" customer.

Why not create different factory methods for each variation?

```
public Customer CreateSilverCustomer(p1, p2) {}
public Customer CreateGoldCustomer{p1, p2, p3) {}
public Customer CreatePlatinumCustomer{p1, p4, platinumOptions) {}
```

```
Public Customer CreateSilverCustomer(p1, p2)
End Sub
Public Customer CreateGoldCustomer(p1, p2, p3)
End Sub
Public Customer CreatePlatinumCustomer(p1, p4, platinumOptions)
End Sub
```

The names express the difference clearly and in business terms. Each method can be tailored to its purpose, as complex or as simple as it needs to be. You can add as many Create methods as you want and add more over time without fear of breaking the core Customer class.

You don't dare do this with different constructor overloads. Constructors can only have one name (the name of the class). Multiple overloads are confusing. You shouldn't change the Customer class everytime you dream up a new flavor of Customer. Keep constructors lean. They should enforce the invariant constraints of the entity and little else.