Contents

You should **write a custom constructor** if any entity properties have required values.

Enforce invariant constraints in constructors

It's a best practice to ensure (to the degree possible) that a new entity is in a valid state. You probably have business rules that stipulate which property values are ***always*** required and what values they must have when they are created. The best place to enforce those rules is in the entity constructor.

You should strive to have a valid entity at all times. You should feel a bit uncomfortable when you can't reach that goal. For example, if a Customer is required always to have a real status code, we might write this constructor:

```
public Customer()
{
  Status = InitialStatus;
}
```

```
Public Sub New()
  Status = InitialStatus
End Sub
```

One sure invariant: every entity must have an **EntityKey**. You should set the EntityKey in the constructor unless it DevForce will be setting it automatically.

Don't initialize every property

Sometimes its better to leave properties alone. The Customer entity presumably has a *Name* property. If the Customer name is allowed to be empty *while in memory*, don't initialize it.

This is a gray area. The customer name probably has a "is required" validation rule. You won't be able to save the customer without a name. You could initialize it in the constructor with a phony name such as "No Name". Technically the customer now passes the name-is-required test. This isn't substantively better than a null or empty string and you risk polluting the database with lots of "No Name" customers.
Most validation rules govern the state of the entity *when it is saved*. When a locally-invalid entity is harmless and there is no satisfactory way to enforce the validity check, let it go.

On the other hand, when the validation rule expresses an invariant constraint - a constraint that must be true at all times - then make sure you set the property value in the constructor.

When to use constructor parameters

Consider writing constructors that take required parameters when essential values can't be determined within the constructor alone.

Imagine this time that a valid customer name is required at all times. It may never be null. The constructor can't possibly know the actual name. Therefore, we require the name to be passed in when creating the customer.

```
public Customer(string name)
{
   EnsureValidName(Name); // throws if null, empty, or bad
  Name = name;
}
```

```
Public Sub New(ByVal name As String)
  EnsureValidName(Name) ' throws if null, empty, or bad
 Name = name
End Sub
```

Ehy do we need "*EnsureValidName*"? Won't the property setter invoke validation and catch a bad name? No it won't. Property setters validate input only when the entity is attached to an EntityManager. This entity is not yet attached to an EntityManager. The *EnsureValidName* method can validate the name using its own VerifierEngine if you wish.

A default parameterless constructor is required

When you add a constructor with parameters, you must also add a default parameterless constructor. DevForce needs that default constructor.

DevForce materializes entities from query results. To "materialize" an entity, DevForce "reconsititutes" it from server data. Sometimes DevForce can reconstitute it through a de-serialization process that by-passes all constructors. In Silverlight, DevForce reconstitutes a queried entity by calling its default, parameterless constructor.

Never assume that your constructor will be called when an entity is reconstituted by query ... or by any other process. DevForce may or may not call the constructor. Don't guess.

In this example, Customer has a default constructor for entity reconstitution and a parameterized constructor for entity creation:

```
internal Customer() {} // required default constructor
public Customer(string name)
{
  EnsureValidName(Name); // throws if null, empty, or bad
 Name = name;
}
```

```
Friend Sub New() ' required default constructor
End Sub
Public Sub New(ByVal name As String)
 EnsureValidName(Name) ' throws if null, empty, or bad
Name = name
End Sub
```

Notice that there is no code inside the default constructor. The application will never call it. It exists only to support reconsitution of an entity during a query. Don't put anything in the default constructor unless you are sure to call it yourself.

Notice also that the default constructor is no longer *public*.

You can hide the default constructor

The implicit default constructor is *public* so DevForce and everyone else can call it. You may not want everyone to call it.

You can hide the default constructor from other assemblies but you may not hide it from DevForce. DevForce uses reflection to find the default constructor. If the entity is to be accessible in Silverlight and you don't want the constructor to be *public*, you can mark *internal*. You can't mark it *private* or *protected* because Silverlight won't let DevForce discover it.

Access modifier constraints are explained elsewhere in connection with entity model definition.

Don't add unnecessary constructor parameters

Minimize the number of constructor parameters and don't let them be optional. Use constructor parameters only for values that absolutely must be passed in from outside.

You can set values for constrained entity properties within the constructor itself. You can set optional properties from the outside, after constructing the instance.