# My Dependency Injection Pattern

## Specification:

### General Concept (What we want to achieve?):

This pattern has two parts:

* Global Factory.
* Globally Accessible Container.

The Factory should be a controller class with access to every other class in the application. Because every single class in the application should have access to the container, then the container cannot have references to those classes.  
The Factory would add all the logic setup to the container and then, optionally, forbidding further registering of new instances.  
Hence, the proposed implementation would be for the container.

The container should offer a way to register instances for different types. It would also implement some logic to allow this registered types to be treated as singleton instances or new instances for every request.

The container’s users would be able to retrieve any registered instances of a given type.

### Features:

1. The container should implement a singleton pattern.
2. Registering instances:
   1. Only references types can be registered.
   2. The concrete type should inherit from the base type.
   3. The instance that the instantiation logic produce should be exactly the specified concrete type.
   4. Register a new concrete type for a base type by providing its instantiation logic.
   5. Multiple instantiation logic can be registered for the same types.
   6. Allow the user to forbid further registering.
      * If a register attempt occur after the user has forbidden further registering it should generate an exception.
   7. Allow the user to reset the container.
      * After resetting the container it would be empty.
      * After resetting the container it would allow the register types (As if the EndRegistering method has never been called before).
3. Retrieving types:
   1. Retrieve all instances for given type.
   2. Retrieve all new instances for a given type.
   3. Retrieve one (any) instance for a given type.
   4. Retrieve one (any) new instance for a given type.
   5. The retrieved instance should belongs to a registered type.
   6. When retrieving one instance, if none such instance has been found a null value should be returned.
   7. The retrieved collection for the “all” request should implement IEnumerable.
   8. When retrieving all the instances, if none such instances has been found and empty collection should be returned.
4. Instances:
   1. The new instances should be created at request time and then returned.
   2. The singleton instances should be created at registered time and returned in every request.

## Design and Implementation:

### Public API:

#### Registering Instances:

The Container would provide the following methods. One method to register the instantiation logic, specifying a Base Type and a Concrete Type:

*   
  Register the instantiation logic for a given Type who inherits from a given Base. The registered type should be the BaseType.  
  The method signature is designed as defensive coding. The constraints would ensures the completion of the features 2.a – 2.c at build time.
*   
  A call to this method should guarantee that no future calls to Register<…>(…) would be allowed.
*   
  A call to this method should guarantee that the state of the container would reset the container.

#### Retrieving Instances:

The Container would provide the following methods for retrieving instances. For each one, if none instances for the given type could be found then the null value should be returned.

The method signature for the “GetAll\*” (return type of IEnumerable<T>) ensures 3.g at build time.

*   
  It will retrieve all instances for the given base type.
*   
  It will retrieve all new instances for a given type.
*   
  It will retrieve the first instance for the given base type.
*   
  It will retrieve the first new instance for the given base type.

### Proposed Internal Structure:

The Sample Implementation is designed to reflect (and test) all the features exposed in this document.

This implementation will implement the Classic Thread-Safe Singleton Pattern. So far we don’t conceive a situation in which a non-singleton container would be needed.

#### Fields:

The container would store its registered types in simple dictionaries, one for the instantiation logic and one for the instances.

*   
  Used to store the provided instance builders: methods that will contains the logic to instantiate a given type. They are stored in order to be able to produce future new instances for those types.
*   
  Used to store specific instances for the application. Those instances would be created when first registering its type hence emulating a singleton behavior.
*   
  Flag value initialized in true, that will worth as a sign that the EndRegistering() has been called.

#### Registering Instances:

When registering a type, the instantiation logic should be executed (and stored) one first time in order to create the singleton version of the registered type.

The container does not need to implement any auxiliar method nor big logic that should be specified.

#### Retrieving Instances:

When retrieving a new instance, the execution logic should be executed, and the created instance should be the one returned.

The container does not need to implement any auxiliary method nor big logic that should be specified. The “GetOne” version would simply take only the first element from the “GetAll” versions.

## Working Example:

  
