using Kore.Utils;

The Utils Package contains stuff used by other packages. Some of the utilities a directly usefull also for other purposes, note that all the Utils are already used by other packages

List of content:

- Scene scoped Singletons (page 2)
- MiniPool (page 4)

Scene scoped Singletons

- To make your class a singleton you just have to inherith the proper template:

```
using Kore.Utils;

public class MyClass:     SceneScopedSingleton< MyClass>
{

}
```

- Now you can access the Singleton Instance like the following:

```
MyClass.Instance.Method();
```

Your class is also a MonoBehaviour now

If you want to access your instance indirectly (through a interface), you have to inherith from SceneScopedSingletonI **(NOTICE THE "I" AT THE END)** template instead.

```
 using Kore.Utils;

public class MyClass:
       SceneScopedSingletonI< MyClass, IMyClass>, IMyClass
{

}
```

In that case, your class have to implement a Interface.

```
 MyClass.Instance.Method();  //Instance now is the "IMyClass" Interface
```

Additional Notes:

Init/Destroy Method: You can perform initialization of your singletons (when they are created the first time) by overriding the Init method. You can also override **OnDestroyCalled()** to perform finalization.

```
public override void Init()
{
    //  Init MyClass here
}
```

Interface for Dependency Injection: The main reason to allow Singletons be exposed through interfaces is to allow your code to be decoupled from Singletons (if you want to do that).

Compare the following:

```
public class AnotherClass
{

   void Method(){
      MyClass.Instance.Method();
   }
}
```

With the following:

```
public class AnotherClass
{
   IMyClass myclass;
   public AnotherClass( IMyClass myclass_interface){
      myclass = myclass_interface; // you can now mock the dependency
   }
   void Method(){
      myclass.Method();
   }
}
```

MiniPool

This class is used as a high-performance object pool. Objects that are pooled needs to implement **IPoolable**: they need to be resetted by the pool so that references are dropped ad object will always be provided in a valid state.

```
public class MyObject: IPoolable
{
	public void Reset() // called by the pool
	{
		// be sure to set to null all references here to help the GC
	}
}
```

You can initialize the pool with a initial size to avoid unnecessary allocations, the poolable amount of objects is limited only by available memory.

```
MiniPool< MyObject> pool = new MiniPool< MyObject>( 100);


MyObject obj = pool.Acquire(); //get a object from the pool
pool.Release( obj); // return the object to the pool
```

Note: after an object have been returned to the pool you should not use it anymore to be sure you will not use it you can set it tu null:

```
MyObject obj = pool.Acquire(); //get a object from the pool
pool.Release( obj); // return the object to the pool
obj = null; // set null to avoid accidental use.
```