**Dependency Injection:**

**Sharing the state** between the **steps** in cucumber using PicoContainer. this means that we must be able to share state between steps.

The glue between Gherkin and the system under test are implemented as regular java methods and implemented in regular java classes. The steps are global in the sense that every step in the same package or sub-package relative to the runner will be found and executed. This allows us to define one step in one class and another step in another class.

When there was only one class, an instance variable or two was probably enough. Now you need to solve the problem with a shared state between the two or more classes with steps.

**Sharing the state between different classes – Dependency injection.**

A **native solution** in java could be share a state using a class with **static fields**, this will work. It is unfortunately, vary for information to leak from one scenario to another. Static field are not cleared while the JVM is running. To clear them, we would either have to reset them manually or restart the JVM, both ways are cumbersome.

The solution in java is to use dependency injection. That is, inject a common object in each class with steps. An object that is recreated every time a new scenario is executed.

Dependency injection can be done in many ways. A simple solution is to **inject dependencies through the constructor** (also called Constructor Dependency Injection).

Cucumber-JVM support many different dependency injection frameworks and Pico-Container is one of the frameworks.

sharing state between steps that depends on the result of an earlier step in the scenario. Sharing state can be done in different ways.

**Singleton class:**

The purpose of the Singleton class is to control object creation limiting the number of objects to only one.

The singleton allows only one entry point to create the new instance of the class.

Since there is only one Singleton instance, any instance fields of a Singleton will occur only once per class, just like static field. Singletons are often useful where you have to control the resources, such as database connections or sockets.

Restrict the instantiation of a class and ensures that only one instance of the class exists in the Java Virtual Machine.

Example code: (Singleton class)

**public class** SingletonDemo {  
  
 *// static class reference as null;* **private static** SingletonDemo *obj* = **null**;  
  
 *// private constructor to restrict instantiation of the class directly* **private** SingletonDemo() {  
 }  
  
 *//* **public static** SingletonDemo createObjectOnce() {  
 **if** (*obj* == **null**) {  
 *obj* = **new** SingletonDemo();  
 }  
 **return** *obj*;  
 }  
  
 *// method of the singleton class* **public void** m1() {  
 System.***out***.println(**"i am method m1 from singleton class"**);  
 }  
*// code to demonstrate objects can created within the same class* **public static void** main(String[] args) {  
 SingletonDemo obj1 = **new** SingletonDemo();  
 obj1.m1();  
 SingletonDemo obj2 = **new** SingletonDemo();  
 obj2.m1();  
 }  
}

Accessing Singleton class

**public class** AccessSingleTon {

*// object cannot be created directly due to private constructor  
// this way it is forced to create object via method where  
// the logic for creating object only once written*

**public static void** main(String[] args) {  
 SingletonDemo s=SingletonDemo.*createObjectOnce*();  
 s.m1();  
 }  
}