|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | |  |  | | --- | --- | | **Design Patterns** |  |      |  |  | | --- | --- | | **Creational Patterns** | | | [**Abstract Factory**](http://www.dofactory.com/Patterns/PatternAbstract.aspx) | Creates an instance of several families of classes | | [**Builder**](http://www.dofactory.com/Patterns/PatternBuilder.aspx) | Separates object construction from its representation | | [**Factory Method**](http://www.dofactory.com/Patterns/PatternFactory.aspx) | Creates an instance of several derived classes | | [**Prototype**](http://www.dofactory.com/Patterns/PatternPrototype.aspx) | A fully initialized instance to be copied or cloned | | [**Singleton**](http://www.dofactory.com/Patterns/PatternSingleton.aspx) | A class of which only a single instance can exist |  |  |  | | --- | --- | | **Structural Patterns** | | | [**Adapter**](http://www.dofactory.com/Patterns/PatternAdapter.aspx) | Match interfaces of different classes | | [**Bridge**](http://www.dofactory.com/Patterns/PatternBridge.aspx) | Separates an object’s interface from its implementation | | [**Composite**](http://www.dofactory.com/Patterns/PatternComposite.aspx) | A tree structure of simple and composite objects | | [**Decorator**](http://www.dofactory.com/Patterns/PatternDecorator.aspx) | Add responsibilities to objects dynamically | | [**Facade**](http://www.dofactory.com/Patterns/PatternFacade.aspx) | A single class that represents an entire subsystem | | [**Flyweight**](http://www.dofactory.com/Patterns/PatternFlyweight.aspx) | A fine-grained instance used for efficient sharing | | [**Proxy**](http://www.dofactory.com/Patterns/PatternProxy.aspx) | An object representing another object |  |  |  | | --- | --- | | **Behavioral Patterns** | | | [**Chain of Resp.**](http://www.dofactory.com/Patterns/PatternChain.aspx) | A way of passing a request between a chain of objects | | [**Command**](http://www.dofactory.com/Patterns/PatternCommand.aspx) | Encapsulate a command request as an object | | [**Interpreter**](http://www.dofactory.com/Patterns/PatternInterpreter.aspx) | A way to include language elements in a program | | [**Iterator**](http://www.dofactory.com/Patterns/PatternIterator.aspx) | Sequentially access the elements of a collection | | [**Mediator**](http://www.dofactory.com/Patterns/PatternMediator.aspx) | Defines simplified communication between classes | | [**Memento**](http://www.dofactory.com/Patterns/PatternMemento.aspx) | Capture and restore an object's internal state | | [**Observer**](http://www.dofactory.com/Patterns/PatternObserver.aspx) | A way of notifying change to a number of classes | | [**State**](http://www.dofactory.com/Patterns/PatternState.aspx) | Alter an object's behavior when its state changes | | [**Strategy**](http://www.dofactory.com/Patterns/PatternStrategy.aspx) | Encapsulates an algorithm inside a class | | [**Template Method**](http://www.dofactory.com/Patterns/PatternTemplate.aspx) | Defer the exact steps of an algorithm to a subclass | | [**Visitor**](http://www.dofactory.com/Patterns/PatternVisitor.aspx) | Defines a new operation to a class without change | |

|  |  |
| --- | --- |
| Name of the Pattern: | Singleton Pattern |
| Objective: | To control number of instances of a class to 1. i.e. Not more than one instance of a class should exist in a single process. |
| Implementation: | The implementation involves following Steps:  1. Make default constructor as Private – to hide it from others to create an instance. 2. Make a static member in the class 3. Make a static public method; return type be class itself |
| Where to use it: |  |
| Category: | Creational Pattern |
| UML Diagram |  |
| ServerSide Code public class Singleton  {  //static member  private static Singleton singleInstance = null;  public int Count { get; set; }  public string Name { get; set; }  //hides the default constructor  private Singleton()  {  Console.WriteLine("Singleton: {0}, Hash:{1}", a.Name == null ? "null " : a.Name, a.GetHashCode());  }  public static Singleton GetInstance()  {  //Checks if the object is not created  if (singleInstance == null)  singleInstance = new Singleton();  return singleInstance;  }  } | |
| ClientSide Implementation: class Program  {  static void Main(string[] args)  {  Singleton a = Singleton.GetInstance();  Console.WriteLine("Singleton: {0}, Hash:{1}", a.Name == null ? "null " : a.Name, a.GetHashCode());  a.Name = "Monali";  Singleton b = Singleton.GetInstance();  Console.WriteLine("Singleton: {0}, Hash:{1}", b.Name, b.GetHashCode());  Singleton c = Singleton.GetInstance();  Console.WriteLine("Singleton: {0}, Hash:{1}", c.Name, c.GetHashCode());  Singleton d = Singleton.GetInstance();  Console.WriteLine("Singleton: {0}, Hash:{1}", d.Name, d.GetHashCode());  a.Count = 15;  Singleton e = Singleton.GetInstance();  Console.WriteLine("Singleton: {0}, Hash:{1}", e.Name, e.GetHashCode());  Console.ReadLine();  }  } | |
| Output:   Same hash code indicates that single instance of the object is being used. | |