**Singleton Design Pattern.**

In [software engineering](https://en.wikipedia.org/wiki/Software_engineering), the **singleton pattern** is a [design pattern](https://en.wikipedia.org/wiki/Design_pattern_(computer_science)) that restricts the [instantiation](https://en.wikipedia.org/wiki/Instantiation_(computer_science)) of a class to one [object](https://en.wikipedia.org/wiki/Object_(computer_science)). This is useful when exactly one object is needed to coordinate actions across the system. The concept is sometimes generalized to systems that operate more efficiently when only one object exists, or that restrict the instantiation to a certain number of objects. The term comes from the [mathematical concept of a singleton](https://en.wikipedia.org/wiki/Singleton_(mathematics)).

### Lazy initialization[[edit](https://en.wikipedia.org/w/index.php?title=Singleton_pattern&action=edit&section=5" \o "Edit section: Lazy initialization)]

This method uses [double-checked locking](https://en.wikipedia.org/wiki/Double_checked_locking_pattern#Usage_in_Java), which should not be used prior to [J2SE 5.0](https://en.wikipedia.org/wiki/Java_Platform,_Standard_Edition), as it is vulnerable to subtle bugs. The problem is that an out-of-order write may allow the instance reference to be returned before the Singleton constructor is executed.[[5]](https://en.wikipedia.org/?title=Singleton_pattern#cite_note-IBM-5)

public class SingletonDemo {

private static SingletonDemo instance = null;

private SingletonDemo() { }

public static synchronized SingletonDemo getInstance() {

if (instance == null) {

instance = new SingletonDemo();

}

return instance;

}

}

### Eager initialization[[edit](https://en.wikipedia.org/w/index.php?title=Singleton_pattern&action=edit&section=6" \o "Edit section: Eager initialization)]

If the program will always need an instance, or if the cost of creating the instance is not too large in terms of time/resources, the programmer can switch to eager initialization, which always creates an instance:

public class Singleton {

private static final Singleton INSTANCE = new Singleton();

private Singleton() {}

public static Singleton getInstance() {

return INSTANCE;

}

}

This method has a number of advantages:

* The static initializer is run when the class is initialized, after class loading but before the class is used by any thread.
* There is no need to synchronize the getInstance() method, meaning all threads will see the same instance and no (expensive) locking is required.
* The final keyword means that the instance cannot be redefined, ensuring that one (and only one) instance ever exists.

### Static block initialization[[edit](https://en.wikipedia.org/w/index.php?title=Singleton_pattern&action=edit&section=7" \o "Edit section: Static block initialization)]

Some authors refer to a similar solution allowing some pre-processing (e.g. for error-checking).[[6]](https://en.wikipedia.org/?title=Singleton_pattern#cite_note-6) In this sense, the traditional approach could be seen as a particular case of this one, as the class loader would do exactly the same processing.

public class Singleton {

private static final Singleton instance;

static {

try {

instance = new Singleton();

} catch (Exception e) {

throw new RuntimeException("Darn, an error occurred!", e);

}

}

public static Singleton getInstance() {

return instance;

}

private Singleton() {

// ...

}

}

### Initialization-on-demand holder idiom[[edit](https://en.wikipedia.org/w/index.php?title=Singleton_pattern&action=edit&section=8" \o "Edit section: Initialization-on-demand holder idiom)]

[University of Maryland](https://en.wikipedia.org/wiki/University_of_Maryland,_College_Park) Computer Science researcher [Bill Pugh](https://en.wikipedia.org/wiki/William_Pugh) has written about the code issues underlying the Singleton pattern when implemented in Java.[[7]](https://en.wikipedia.org/?title=Singleton_pattern#cite_note-7) Pugh's efforts on the "[Double-checked locking](https://en.wikipedia.org/wiki/Double-checked_locking)" idiom led to changes in the Java memory model in Java 5 and to what is generally regarded as the standard method to implement Singletons in Java. The technique known as the [initialization-on-demand holder idiom](https://en.wikipedia.org/wiki/Initialization-on-demand_holder_idiom) is as lazy as possible, and works in all known versions of Java. It takes advantage of language guarantees about class initialization, and will therefore work correctly in all Java-compliant compilers and virtual machines.

The nested class is referenced no earlier (and therefore loaded no earlier by the class loader) than the moment that getInstance() is called. Thus, this solution is [thread-safe](https://en.wikipedia.org/wiki/Thread_safety) without requiring special language constructs (*i.e.* volatile or synchronized).

public class Singleton {

// Private constructor. Prevents instantiation from other classes.

private Singleton() { }

/\*\*

\* Initializes singleton.

\*

\* {@link SingletonHolder} is loaded on the first execution of {@link Singleton#getInstance()} or the first access to

\* {@link SingletonHolder#INSTANCE}, not before.

\*/

private static class SingletonHolder {

private static final Singleton INSTANCE = new Singleton();

}

public static Singleton getInstance() {

return SingletonHolder.INSTANCE;

}

}

**FactoryDesign Pattern.**

In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

From wisloadconfig.xml we read all the data sources and stores the url, id, password in one object and loads the driver. And it will store that object in Hashmap with key as datasource name and value as dataconnectionfactory object.

Now while connecting data base c=DataSourceConnectionFactory.*getFactory*("warrantyData").getConnection();

getFactory method will return the connectionManager object for an datasource name and it will make the connection.

We have ConnectionManger interface which has the getConnection() and releaseConnection () methods and ConnectionManagerBasic will implement the interface ConnectionManger, so while connecting the database I will give the datasource name and from that it will get the ConnectionManager object for that datasource and I will call the getConnection() method.

So here I am creating the new object without exposing the logic how to create the object. Here we are passing the datasource name for getFactory method of DataSourceConnectionFactory class and it will return the ConnectionManager object for that particular.

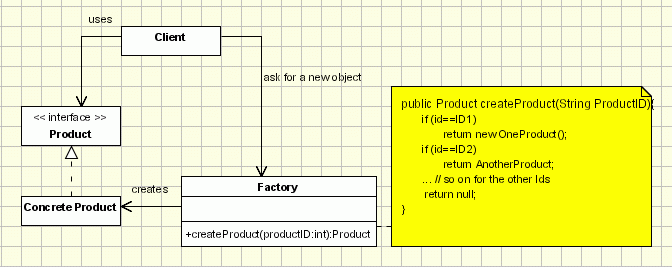
And for the connectionManger object we are using the Singleton Design pattern, i.e, it will create the single object per JVM Instance.

**Common Basic Example:**

1. Creates objects without exposing the instantiation logic to the client.
2. refers to the newly created object through a common interface

**Implementation:**

* The client needs a product, but instead of creating it directly using the new operator, it asks the factory object for a new product, providing the information about the type of object it needs.
* The factory instantiates a new concrete product and then returns to the client the newly created product (casted to abstract product class).
* The client uses the products as abstract products without being aware about their concrete implementation.



Now if there two Product Classes say Product1, Product2 which are implementing the Product interface and now client is asking for Product1 object to factory class and factory class providing the Product1 object basis on the some ID. Now if want to add another new product, so I will create Product3 class which implements Product Interface, to get its object I need to add only two line of code in createProduct method its reduce the code duplication.

The key advantages of the Factory pattern are twofold:

1. The places that need an implementation of the product do not need to know how to construct one. The factory holds that information.

Do you want to know what arguments to pass to the particular constructor? Or what dependencies you have to inject? Or how to register the implementation class with the database after it is fully configured? No? Let the factory look after all that stuff.

1. The places that need an implementation of the product do not need to know at the time of module description (i.e., at compilation time) what the name of the implementation class is.

Thus, a need not have anything to do with A; the “what to build” can be described in terms of the desired non-functional properties, and not just the name. This is much more flexible.

When you don't use the factory, you would have that code repeatedly used in several locations in your code.

**Reuse.** If I want to instantiate in many places, I don't have to repeat my condition, so when I come to add a new class, I don't run the risk of missing one.

**Unit-Testability.** I can write 3 tests for the factory, to make sure it returns the correct types on the correct conditions, then my calling class only needs to be tested to see if it calls the factory and then the required methods on the returned class. It needs to know nothing about the implementation of the factory itself or the concrete classes.

**Extensibility.** When someone decides we need to add a new class D to this factory, none of the calling code, neither unit tests nor implementation, ever needs to be told. We simply create a new class D and extend our factory method. This is the very definition of [Open-Closed Principle](http://www.oodesign.com/open-close-principle.html).

**Command Design Pattern.**

Command design pattern is used to encapsulate a request as an object and pass to an invoker, wherein the invoker does not knows how to service the request but uses the encapsulated command to perform an action.

To understand command design pattern we should understand the associated key terms like client, command, command implementation, invoker, receiver.

* Command is an interface with execute method. It is the core of contract.
* A client creates an instance of a command implementation and associates it with a receiver.
* An invoker instructs the command to perform an action.
* A Command implementation’s instance creates a binding between the receiver and an action.
* Receiver is the object that knows the actual steps to perform the action.

In [object-oriented programming](https://en.wikipedia.org/wiki/Object-oriented_programming), the **command pattern** is a [behavioral](https://en.wikipedia.org/wiki/Behavioural_Pattern) [design pattern](https://en.wikipedia.org/wiki/Design_pattern_(computer_science)) in which an object is used to [encapsulate](https://en.wikipedia.org/wiki/Information_Hiding) all information needed to perform an action or trigger an event at a later time. This information includes the method name, the object that owns the method and values for the method parameters.

Four terms always associated with the command pattern are *command*, *receiver*, *invoker* and *client*. A *command* object knows about *receiver* and invokes a method of the receiver. Values for parameters of the receiver method are stored in the command. The *receiver* then does the work. An *invoker* object knows how to execute a command, and optionally does bookkeeping about the command execution. The invoker does not know anything about a concrete command, it knows only about command interface. Both an invoker object and several command objects are held by a *client* object. The client decides which commands to execute at which points. To execute a command, it passes the command object to the invoker object.Using command objects makes it easier to construct general components that need to delegate, sequence or execute method calls at a time of their choosing without the need to know the class of the method or the method parameters. Using an invoker object allows bookkeeping about command executions to be conveniently performed, as well as implementing different modes for commands, which are managed by the invoker object, without the need for the client to be aware of the existence of bookkeeping or modes.