**DESIGN PATTERN**

1. SINGLETON DESIGN PATERN
2. FACTORY DESIGN PATERN
3. BUILDER DESIGN PATERN
4. MICROSERVICES DESIGN PATERN

# SINGLETON DESIGN PATERN

* Singleton pattern ***restricts the instantiation of a class and ensures that only one instance of the class exists in the java virtual machine.***
* The singleton class **must provide a global access point to get the instance of the class**.
* Singleton pattern is used for [logging](https://www.journaldev.com/977/logger-in-java-logging-example), drivers objects, caching and [thread pool](https://www.journaldev.com/1069/threadpoolexecutor-java-thread-pool-example-executorservice).
* Singleton design pattern is also used in other design patterns like [Abstract Factory](https://www.journaldev.com/1418/abstract-factory-design-pattern-in-java), [Builder](https://www.journaldev.com/1425/builder-design-pattern-in-java), [Prototype](https://www.journaldev.com/1440/prototype-design-pattern-in-java), [Facade](https://www.journaldev.com/1557/facade-design-pattern-in-java) etc.
* Singleton design pattern is used in core java classes also, for example java.lang.Runtime, java.awt.Desktop.

To implement a Singleton pattern, we have different approaches but all of them have the following common concepts.

* **Private constructor** to restrict instantiation of the class from other classes.
* **Private static variable** of the same class that is the only instance of the class.
* **Public static method** that returns the instance of the class, this is the global access point for outer world to get the instance of the singleton class.

In further sections, we will learn different approaches of Singleton pattern implementation and design concerns with the implementation.

1. **[Eager initialization](https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples" \l "eager-initialization)**

In eager initialization, the instance of Singleton Class is created at the time of class loading, this is the easiest method to create a singleton class.

**public** **class** SingletoneDPClass {

**//eager initialization**

**public** **static** **final** SingletoneDPClass ***eagerInit*** = **new** SingletoneDPClass();

// private constructor to avoid client applications to use constructor

**private** SingletoneDPClass() {

}

}

Drawback:

1. instance is created even though client application might not be using it.
2. doesn’t provide any options for exception handling
3. **[Static block initialization](https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples" \l "static-block-initialization)**

It Is just like eager initialization only we are creating class object in static block so that we can hadle exceptions also.

**public** **class** SingletoneDPClass {

**private** **static** SingletoneDPClass *instance*;

// private constructor to avoid client applications to use constructor

**private** SingletoneDPClass() {

}

// static block initialization for exception handling

**static** {

**try** {

*instance* = **new** SingletoneDPClass();

} **catch** (Exception e) {

**throw** **new** RuntimeException("Exception occured in creating singleton instance");

}

}

**public** **static** SingletoneDPClass getInstance() {

**return** *instance*;

}

}

Drawback:

Both eager initialization and static block initialization creates the instance even before it’s being used and that is not the best practice to use. So in further sections, we will learn how to create a Singleton class that supports lazy initialization.

1. **[Lazy Initialization](https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples" \l "lazy-initialization)**

Here we will be sending an object based on method call.

**public** **class** SingletoneDPClass {

**private** **static** SingletoneDPClass *instance*;

// private constructor to avoid client applications to use constructor

**private** SingletoneDPClass() {

}

**public** **static** SingletoneDPClass getInstance() {

**if** (*instance* == **null**)

**return** **new** SingletoneDPClass();

**else**

**return** *instance*;

}

}

Drawback:

Above method is not thread safe. In multi threaded environment, multiple thread can execute the method to create many objects. So we will go for thread safe approach.

1. **[Thread Safe Singleton](https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples" \l "thread-safe-singleton)**

For achieving thread safe singletone class, we will make ***getInstance() synchronized.*** *But* it reduces the performance because of the cost associated with the synchronized method, although we need it only for the first few threads who might create the separate instances.

Here ***double checked locking***principle is used. In this approach, the synchronized block is used inside the if condition with an additional check to ensure that only one instance of a singleton class is created.

**public** **class** SingletoneDPClass {

**private** **static** SingletoneDPClass *instance*;

// private constructor to avoid client applications to use constructor

**private** SingletoneDPClass() {

}

**public** **static** SingletoneDPClass getInstance() {

**if** (*instance* == **null**) {

**synchronized (SingletoneDPClass.class) {**

**if (*instance* == null)**

***instance* = new SingletoneDPClass();**

**}**

}

**return** *instance*;

}

}

1. [Bill Pugh Singleton Implementation](https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples" \l "bill-pugh-singleton)
2. [Using Reflection to destroy Singleton Pattern](https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples" \l "reflection-and-singleton)
3. [Serialization and Singleton](https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples" \l "serialization-and-singleton)

When we serialize any ibject that object is transfered to file or network. When we de-serialize then we get new instance of an object. To prevent this, we just need to override **readResolve()** method in the class so that it will not create new object.

// implement readResolve method

**protected** Object readResolve() {

**return** *test*;

}

1. Prevent from cloning

So when we clone any object we can get exact copy of original object. If we are calling clone() on singleton object, then we might get newly created instance. To avoid it, **we need to override clone method and throw CloneNotSupportedException from that method.**

Preventing singleton violations from

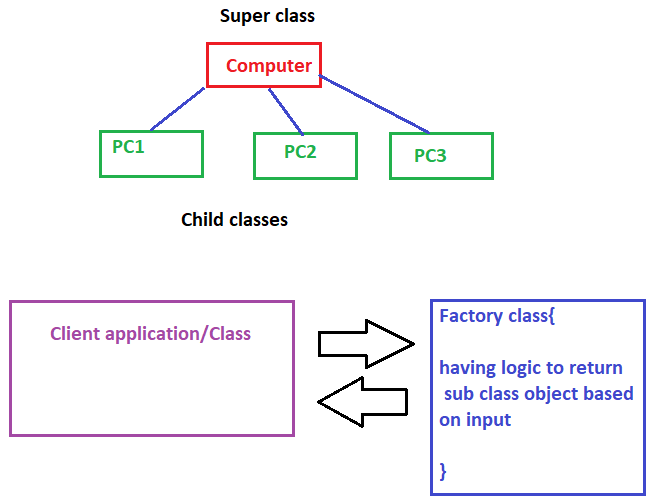
Clonning,

Reflectipn API

# FACTORY DESIGN PATERN

The factory design pattern is used ***when we have a superclass with multiple sub-classes and based on input, we need to return one of the sub-class.***

Suppose we have one superclass Computer and multiple subclasses like PC1, PC2, PC3. So based on the input we are sending we should get subclass object from a spate class called as factoryclass.



**public** **interface** **Computer** {

**public** String getRam();

**public** String getHD();

}

**public** **class** PC1 **implements** Computer {

**private** String ram;

**private** String hd;

**public** PC1(String ram, String hd) {

**super**();

**this**.ram = ram;

**this**.hd = hd;

}

@Override

**public** String getRam() {

**return** **this**.ram;

}

@Override

**public** String getHD() {

**return** **this**.hd;

}

@Override

**public** String toString() {

**return** "PC1 [ram=" + ram + ", hd=" + hd + "]";

}

}

**public** **class** PC2 **implements** Computer {

**private** String ram;

**private** String hd;

**public** PC2(String ram, String hd) {

**super**();

**this**.ram = ram;

**this**.hd = hd;

}

@Override

**public** String getRam() {

**return** **this**.ram;

}

@Override

**public** String getHD() {

**return** **this**.hd;

}

@Override

**public** String toString() {

**return** "PC2 [ram=" + ram + ", hd=" + hd + "]";

}

}

**public** **class** PC3 **implements** Computer {

**private** String ram;

**private** String hd;

**public** PC3(String ram, String hd) {

**super**();

**this**.ram = ram;

**this**.hd = hd;

}

@Override

**public** String getRam() {

**return** **this**.ram;

}

@Override

**public** String getHD() {

**return** **this**.hd;

}

@Override

**public** String toString() {

**return** "PC3 [ram=" + ram + ", hd=" + hd + "]";

}

}

**Factory class**

**public** **class** FactoryClass {

**public** **static** Computer getInstance(String input) {

**if** (input.equalsIgnoreCase("PC1"))

**return** **new** PC1("8GB", "Sandisk");

**else** **if** (input.equalsIgnoreCase("PC2"))

**return** **new** PC2("4GB", "Toshiba");

**else** **if** (input.equalsIgnoreCase("PC3"))

**return** **new** PC3("4GB", "HP");

**else**

**return** **null**;

}

}

**Client application**

**public** **class** ClientApplication {

**public** **static** **void** main(String[] args) {

FactoryClass fc = **new** FactoryClass();

Computer computer = fc.*getInstance*("pc1");

System.***out***.println(computer);

}

}

**Builder Design Pattern**

Factory design pattern has some drawback given below

1. Too Many arguments to pass from client program to the Factory class that can be error prone because most of the time, the type of arguments are same and from client side its hard to maintain the order of the argument.
2. Some of the parameters might be optional but in Factory pattern, we are forced to send all the parameters and optional parameters need to send as NULL.
3. If the object is heavy and its creation is complex, then all that complexity will be part of Factory classes that is confusing.