**Builder Pattern**

For example we have class person

**public** **class** Person {

**public** String name ;

**public** **int** height ;

**public** String color ;

**public** **int** age ;

**public** String address ;

}

If we need to create constructor for this class we need to add all the parameters in constructor.

**public** Person(String name , **int** height , String color ,

**int** age ,String address) {

// **TODO** Auto-generated constructor stub

}

So when we need to create object we need to pass all the values.

If we don’t need to pass all the values but you need to create object for this person class, then we need to add multiple constructors as shown below.

**public** Person(String name , **int** height , String color ,

**int** age ,String address) {

// **TODO** Auto-generated constructor stub

}

**public** Person(String name) {

// **TODO** Auto-generated constructor stub

}

**public** Person(String name , **int** height ) {

// **TODO** Auto-generated constructor stub

}

**public** Person(String name , **int** height , String color

) {

// **TODO** Auto-generated constructor stub

}

This approach will ends up having large number of constructors for different combinations of variables.

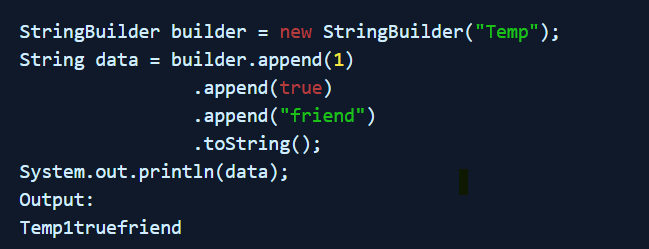
We can create those many constructors but class will become too large and difficult to understand.

Builder pattern will solve this problem.

This should be used only when you want to build different immutable objects using same object building process.

See the implementation in code.

Example for builderpattern.



**Singleton Pattern.**

When we need to create only one object for class we need to use singleton.

When to use?

1. Database connections.
2. File managers.

Different ways to create singletonobject.

1. Singleton with Eager Initialization
2. Singleton with Lazy initialization NoThreads
3. Singleton with synchronized ThreadSafe
4. Singleton with synchronized block
5. Singleton with static block
6. Singleton thread safe with double check

See the implementation of each way in code.

**Prototype Pattern.**

The Prototype pattern is generally used when we have an instance of the class (prototype) and we'd like to create new objects by just copying the prototype.

In some games, we want trees or buildings in the background. We may realize that we don't have to create new trees or buildings and render them on the screen every time the character moves.

So, we create an instance of the tree first. Then we can create as many trees as we want from this instance (prototype) and update their positions. We may also choose to change the color of the trees for a new level in the game.

This approach saves costly resources and time, especially when the object creation is a heavy process.

One of the ways we can implement this pattern in Java is by using the *clone()* method. To do this, we'd implement the *Cloneable* interface

See the implementation in code.

**Factory Pattern:**

The factory design pattern says that define an interface ( A java interface or an abstract class) and let the subclasses decide which object to instantiate.

If the class don’t know which object need to be created, then we will use factory pattern.

In factory pattern we have factory method which decides the respective object.

See the implementation in code.

**Abstract Factory Pattern:**

* The abstract factory pattern is a creational pattern which provides a way to encapsulate a group of individual factories that have a common purpose or theme; without specifying their concrete classes.
* The Abstract Factory patterns work around a super-factory which creates other factories.
* This super-factory is also known as a factory of factories.

See the implementation in code.