**1. Builder Design Pattern:**

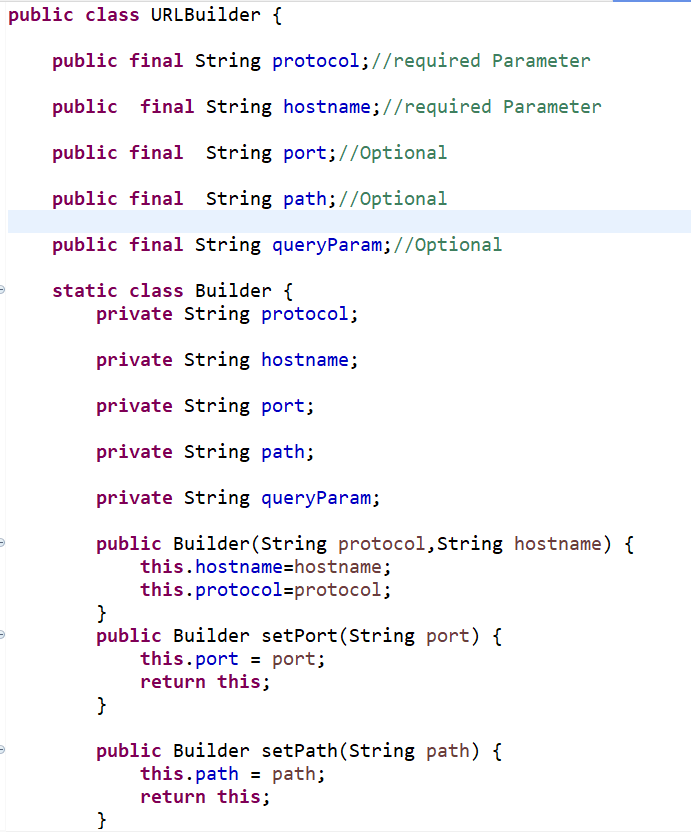
a) **Builder** is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

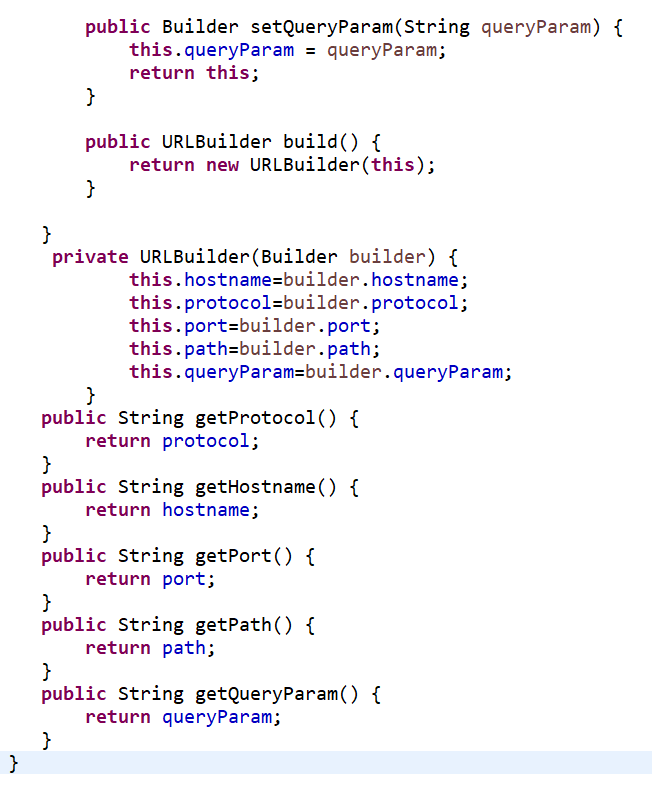
b) The Builder pattern suggests that you extract the object construction code out of its own class and move it to separate objects called builders.

c) A **Builder Pattern** solves the issue with many optional parameters and inconsistent states by providing a way to build the object step-by-step and provide a method that will return the final Object.

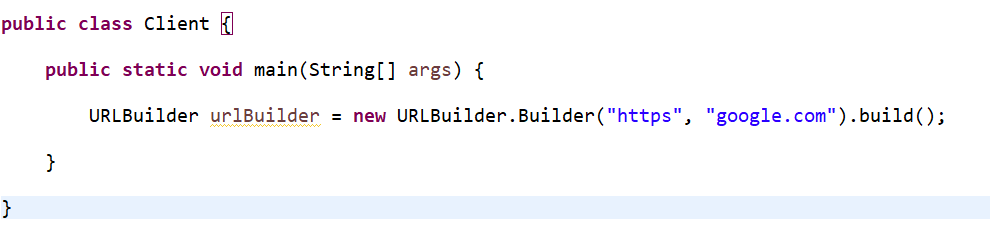
d) In Java StringBuilder and StringBuffer class uses Builder Pattern.

Code:





ClientCode:



**2. Factory Method Pattern:**

a) **Factory Method** is a creational design pattern that provides an interface/abstract class for creating objects in a superclass but allows subclasses to alter the type of objects that will be created.

Note: Superclass in factory design pattern can be an interface, [abstract class](https://www.digitalocean.com/community/tutorials/abstract-class-in-java) or a normal java class.

b) 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. This pattern takes out the responsibility of the instantiation of a class from the client program to the factory class.

c) This pattern is particularly useful when the exact types of objects to be created may vary or need to be determined at runtime, enabling flexibility and extensibility in object creation. It helps in promoting loose coupling by eliminating the need to bind application-specific classes into the code.

Code:

// Library classes

abstract class Vehicle {

public abstract void printVehicle();

}

class TwoWheeler extends Vehicle {

public void printVehicle() {

System.out.println("I am two-wheeler");

}

}

class FourWheeler extends Vehicle {

public void printVehicle() {

System.out.println("I am four-wheeler");

}

}

// Factory Interface

interface VehicleFactory {

Vehicle createVehicle();

}

// Concrete Factory for TwoWheeler

class TwoWheelerFactory implements VehicleFactory {

public Vehicle createVehicle() {

return new TwoWheeler();

}

}

// Concrete Factory for FourWheeler

class FourWheelerFactory implements VehicleFactory {

public Vehicle createVehicle() {

return new FourWheeler();

}

}

// Client class

class Client {

private Vehicle vehicle;

public Client(VehicleFactory factory) {

vehicle = factory.createVehicle();

}

public Vehicle getVehicle() {

return vehicle;

}

}

// Driver program

public class Driver {

public static void main(String[] args) {

VehicleFactory twoWheelerFactory = new TwoWheelerFactory();

Client twoWheelerClient = new Client(twoWheelerFactory);

Vehicle twoWheeler = twoWheelerClient.getVehicle();

twoWheeler.printVehicle();

}

}