**Factory Method Pattern**

**Factory Method Pattern**

The factory method design pattern abstract the process of object creation and allows the object to be created at run-time when it is required which means it Define an interface for creating an object, but let subclasses decide which class to instantiate.

Lets us understand with an example.

In Below example,Client creates objects of either TwoWheeler or FourWheeler based on input during constructing its object.

If library class introduces a new class ThreeWheeler to incorporate three wheeler vehicles. Now client will end up chaining a new else if in the conditional ladder to create objects of ThreeWheeler, Which in turn will need Client to be recompiled. So, each time a new change is made at the library side, Client would need to make some corresponding changes at its end and recompile the code. This is considered to be a very bad practice of design.

#include <iostream>

using namespace std;

// Library classes

class Vehicle {

public:

virtual void printVehicle() = 0;

};

class TwoWheeler : public Vehicle {

public:

void printVehicle() {

cout << "I am two wheeler" << endl;

}

};

class FourWheeler : public Vehicle {

public:

void printVehicle() {

cout << "I am four wheeler" << endl;

}

};

// Client (or user) class

class Client {

public:

Client(int type) {

// Client explicitly creates classes according to type

if (type == 1)

pVehicle = new TwoWheeler();

else if (type == 2)

pVehicle = new FourWheeler();

else

pVehicle = NULL;

}

~Client() {

if (pVehicle)

{

delete[] pVehicle;

pVehicle = NULL;

}

}

Vehicle\* getVehicle() {

return pVehicle;

}

private:

Vehicle \*pVehicle;

};

// Driver program

int main() {

Client \*pClient = new Client(1);

Vehicle \* pVehicle = pClient->getVehicle();

pVehicle->printVehicle();

return 0;

}

We can solve the above issues using the factory Method pattern.

**Implementation**

Elements of factor method are.

**Product** (In our example its Vehicle Class)

* Objects created by factory method

**Concrete Product** (In our example its TwoWheeler ..)

* Implements the interface

**Creator (In our example its Client class)**

* Declares the factory method
* Calls the factory method

**Concrete Creator (Newly introduced )**

* Overwrites the factory method

#include <iostream>

using namespace std;

// Library classes **(Product)**

class Vehicle {

public:

virtual void printVehicle() = 0;

};

class TwoWheeler : public Vehicle { //( **Concrete Product)**

public:

void printVehicle() {

cout << "I am two wheeler" << endl;

}

};

class FourWheeler : public Vehicle { //( **Concrete Product)**

public:

void printVehicle() {

cout << "I am four wheeler" << endl;

}

};

// Client (or user) class **Creator**

class Client {

public:

**virtual** Vehicle \* FactoryMethod() **const** = 0;

~Client(){};

void printVehicle() {

Vehicle \* vehicle = **this**->FactoryMethod();

Vehicle.printVehicle();

}

private:

Vehicle \*pVehicle;

};

**class** **ConcreteCreator1** : **public** Client {

**public**:

Product\* FactoryMethod() **const** **override** {

**return** **new** TwoWheeler ();

}

};

**class** **ConcreteCreator2** : **public** Client {

**public**:

Product\* FactoryMethod() **const** **override** {

**return** **new** ConcreteProduct2();

}

};

// Driver program

int main() {

Creator\* creator = **new** ConcreteCreator1();

creator.printVehicle();

return 0;

}