**Abstract Factory Pattern**

**Introduction**

Abstract Factory design pattern is one of the Creational pattern. Abstract Factory pattern is almost similar to [Factory Pattern](https://www.geeksforgeeks.org/design-patterns-set-2-factory-method/) is considered as another layer of abstraction over factory pattern. Abstract Factory patterns work around a super-factory which creates other factories.  
Abstract factory pattern implementation provides us with a framework that allows us to create objects that follow a general pattern. So at runtime, the abstract factory is coupled with any desired concrete factory which can create objects of the desired type.  
**Let see the GOFs representation of Abstract Factory Pattern :** 

https://media.geeksforgeeks.org/wp-content/uploads/AbstractFactoryPattern-2.png

*UML class diagram example for the Abstract Factory Design Pattern.* 

* **AbstractFactory**: Declares an interface for operations that create abstract product objects.
* **ConcreteFactory**: Implements the operations declared in the AbstractFactory to create concrete product objects.
* **Product**: Defines a product object to be created by the corresponding concrete factory and implements the AbstractProduct interface.
* **Client**: Uses only interfaces declared by AbstractFactory and AbstractProduct classes.

Abstract Factory provides interfaces for creating families of related or dependent objects without specifying their concrete classes.  
Client software creates a concrete implementation of the abstract factory and then uses the generic interfaces to create the concrete objects that are part of the family of objects.   
The client does not know or care which concrete objects it gets from each of these concrete factories since it uses only the generic interfaces of their products.  
So with this idea of Abstract Factory pattern, we will now try to create a design that will facilitate the creation of related objects. 

**Implementation**

Let’s take an example, Suppose we want to build a global car factory. If it was a [factory design pattern](https://www.geeksforgeeks.org/design-patterns-set-2-factory-method/), then it was suitable for a single location. But for this pattern, we need multiple locations and some critical design changes.  
We need car factories in each location like IndiaCarFactory, USACarFactory, and DefaultCarFactory. Now, our application should be smart enough to identify the location where it is being used, so we should be able to use the appropriate car factory without even knowing which car factory implementation will be used internally. This also saves us from someone calling the wrong factory for a particular location.  
Here we need another layer of abstraction that will identify the location and internally use correct car factory implementation without even giving a single hint to the user. This is exactly the problem, which an abstract factory pattern is used to solve.

* Java

|  |
| --- |
| // Java Program to demonstrate the  // working of Abstract Factory Pattern    enum CarType  {      MICRO, MINI, LUXURY  }    abstract class Car  {      Car(CarType model, Location location)      {          this.model = model;          this.location = location;      }        abstract void construct();        CarType model = null;      Location location = null;        CarType getModel()      {          return model;      }        void setModel(CarType model)      {          this.model = model;      }        Location getLocation()      {          return location;      }        void setLocation(Location location)      {          this.location = location;      }        @Override      public String toString()      {          return "CarModel - "+model + " located in "+location;      }  }    class LuxuryCar extends Car  {      LuxuryCar(Location location)      {          super(CarType.LUXURY, location);          construct();      }      @Override      protected void construct()      {          System.out.println("Connecting to luxury car");      }  }    class MicroCar extends Car  {      MicroCar(Location location)      {          super(CarType.MICRO, location);          construct();      }      @Override      protected void construct()      {          System.out.println("Connecting to Micro Car ");      }  }    class MiniCar extends Car  {      MiniCar(Location location)      {          super(CarType.MINI,location );          construct();      }        @Override      void construct()      {          System.out.println("Connecting to Mini car");      }  }    enum Location  {    DEFAULT, USA, INDIA  }    class INDIACarFactory  {      static Car buildCar(CarType model)      {          Car car = null;          switch (model)          {              case MICRO:                  car = new MicroCar(Location.INDIA);                  break;                case MINI:                  car = new MiniCar(Location.INDIA);                  break;                case LUXURY:                  car = new LuxuryCar(Location.INDIA);                  break;                    default:                  break;            }          return car;      }  }    class DefaultCarFactory  {      public static Car buildCar(CarType model)      {          Car car = null;          switch (model)          {              case MICRO:                  car = new MicroCar(Location.DEFAULT);                  break;                case MINI:                  car = new MiniCar(Location.DEFAULT);                  break;                case LUXURY:                  car = new LuxuryCar(Location.DEFAULT);                  break;                    default:                  break;            }          return car;      }  }      class USACarFactory  {      public static Car buildCar(CarType model)      {          Car car = null;          switch (model)          {              case MICRO:                  car = new MicroCar(Location.USA);                  break;                case MINI:                  car = new MiniCar(Location.USA);                  break;                case LUXURY:                  car = new LuxuryCar(Location.USA);                  break;                    default:                  break;            }          return car;      }  }        class CarFactory  {      private CarFactory()      {        }      public static Car buildCar(CarType type)      {          Car car = null;          // We can add any GPS Function here which          // read location property somewhere from configuration          // and use location specific car factory          // Currently I'm just using INDIA as Location          Location location = Location.INDIA;            switch(location)          {              case USA:                  car = USACarFactory.buildCar(type);                  break;                case INDIA:                  car = INDIACarFactory.buildCar(type);                  break;                default:                  car = DefaultCarFactory.buildCar(type);            }            return car;        }  }    class AbstractDesign  {      public static void main(String[] args)      {          System.out.println(CarFactory.buildCar(CarType.MICRO));          System.out.println(CarFactory.buildCar(CarType.MINI));          System.out.println(CarFactory.buildCar(CarType.LUXURY));      }  } |

Output : 

Connecting to Micro Car

CarModel - MICRO located in INDIA

Connecting to Mini car

CarModel - MINI located in INDIA

Connecting to luxury car

CarModel - LUXURY located in INDIA

**Difference**

* The main difference between a “factory method” and an “abstract factory” is that the factory method is a single method, and an abstract factory is an object.
* The factory method is just a method, it can be overridden in a subclass, whereas the abstract factory is an object that has multiple factory methods on it.
* The Factory Method pattern uses inheritance and relies on a subclass to handle the desired object instantiation.

**Advantages:**This pattern is particularly useful when the client doesn’t know exactly what type to create.

* **Isolation of concrete classes:** The Abstract Factory pattern helps you control the classes of objects that an application creates. Because a factory encapsulates the responsibility and the process of creating product objects, it isolates clients from implementation classes. Clients manipulate instances through their abstract interfaces. Product class names are isolated in the implementation of the concrete factory; they do not appear in client code.
* **Exchanging Product Families easily:** The class of a concrete factory appears only once in an application, that is where it’s instantiated. This makes it easy to change the concrete factory an application uses. It can use various product configurations simply by changing the concrete factory. Because an abstract factory creates a complete family of products, the whole product family changes at once.
* **Promoting consistency among products:** When product objects in a family are designed to work together, it’s important that an application use objects from only one family at a time. AbstractFactory makes this easy to enforce.n.

**Disadvantages**

* **Difficult to support new kinds of products:** Extending abstract factories to produce new kinds of Products isn’t easy. That’s because the AbstractFactory interface fixes the set of products that can be created. Supporting new kinds of products requires extending the factory interface, which involves changing the AbstractFactory class and all of its subclasses.