

# Abstract Factory Design Pattern

- one of the most commonly used design pattern
- part of creational design pattern
- uses in almost all the libraries in JDK, Spring framework.

define an interface ( A java interface or an abstract class) for creating object and let the subclasses decide which class to instantiate.

- promotes loose-coupling by eliminating the need to bind application-specific classes to the code

## When to use Abstract Design pattern?

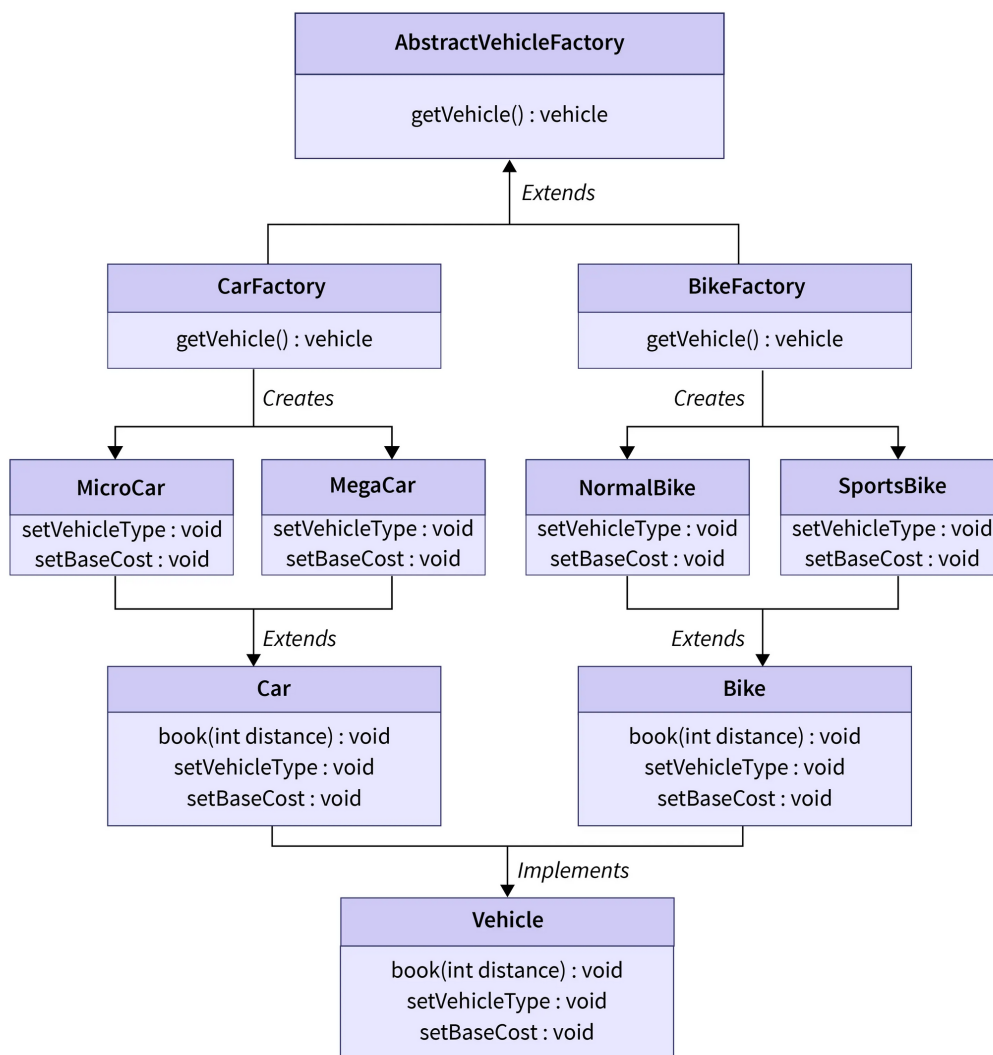
- objects does not need details of how classes are created and represented
- system need to operate with one of several families of products
- system needs to be configured with one of a multiple

# Abstract Factory Design Pattern

family of objects.

- provide a library and want to show just the interface, not implementation of the library components.

## Class Diagram example



# Abstract Factory Design Pattern

## Pseudo code for the example:

1) interface Vehicle

```
book(int distance)
setVehicleType()
setBaseCost()
setVehicleChargesPerUnitDistance()
int calculateCostOfBooking(int distance)
```

define generic vehicle

2) abstract class Car implements Vehicle

```
String carType
int baseCost
int chargesPerUnitDistance
```

define abstract class Car that  
implement Vehicle

```
book(int distance){
    setVehicleType()
    setBaseCost()
    setVehicleChargesPerUnitDistance()
    int cost = calculateCostOfBooking(distance)
    print(carType + ", " + distance + ", " + cost + ". ")
}
```

```
int calculateCostOfBooking(int distance){
    int serviceCharge = 3
    return baseCost + chargesPerUnitDistance * distance + serviceCharge
}
```

3) abstract class Bike implements Vehicle

```
String bikeType
int baseCost
int chargesPerUnitDistance
```

```
book(int distance)
    setVehicleType()
    setBaseCost()
    setVehicleChargesPerUnitDistance()
    int cost = calculateCostOfBooking(distance)
    print(bikeType + ", " + distance + ", " + cost + ". ")
}
```

define abstract class Bike that  
implement Vehicle

# Abstract Factory Design Pattern

```
int calculateCostOfBooking(int distance){  
    int serviceCharge = 3  
    return baseCost + chargesPerUnitDistance * distance + serviceCharge  
}
```

4) class MicroCar extends Car

```
MicroCar()
```

```
setVehicleType()  
    carType = "Micro"
```

```
setBaseCost()  
    baseCost = 50
```

```
setVehicleChargesPerUnitDistance()  
    chargesPerUnitDistance = 10
```

define concrete class MicroCar  
for Car

5) class SportsBike extends Bike

```
PersonalAuto()
```

```
setVehicleType()  
    bikeType = "Sports"
```

```
setBaseCost()  
    baseCost = 10
```

```
setVehicleChargesPerUnitDistance()  
    chargesPerUnitDistance = 15
```

define concrete class SportsBike  
for Bike

6) abstract class AbstractVehicleFactory  
 abstract Vehicle getVehicle(String type)

define abstract class  
AbstractVehicleFactory which  
is factory of factory

# Abstract Factory Design Pattern

7) class CarFactory extends AbstractVehicleFactory

```
Vehicle getVehicle(String type)
    if(type.equalsIgnoreCase("Micro"))
        return new MicroCar()
    else if(type.equalsIgnoreCase("Mini"))
        return new MiniCar()
    else if(type.equalsIgnoreCase("Mega"))
        return new MegaCar()
    else
        return new MiniCar()
```

define concrete class  
CarFactory that extends  
AbstractVehicleFactory

8) class BikeFactory extends AbstractVehicleFactory

```
Vehicle getVehicle(String type)
    if(type.equalsIgnoreCase("Sports"))
        return new SportsBike()
    else if(type.equalsIgnoreCase("Normal"))
        return new NormalBike()
    else
        return new NormalBike()
```

define concrete class  
BikeFactory that extends  
AbstractBikeFactory

9) class FactoryProvider

```
static AbstractVehicleFactory getVehicleFactory(String factoryType)
    if(factoryType.equalsIgnoreCase("Car"))
        return new CarFactory()
    else if(factoryType.equalsIgnoreCase("Auto"))
        return new AutoFactory()
    else if(factoryType.equalsIgnoreCase("Bike"))
        return new BikeFactory()
    else
        return new CarFactory()
```

define FactoryProvider that  
provides required vehicle factory

# Abstract Factory Design Pattern

10) class AbstractFactoryPatternDemoClient

main()

int distance = 10

Client class

/\*

\* Book a Micro Car for a distance of 10 kms

\*/

AbstractVehicleFactory carFactory = FactoryProvider.getVehicleFactory("Car")

Vehicle miniCar = carFactory.getVehicle("Micro")

miniCar.book(distance)

/\*

\* Book a Personal Auto for a distance of 10 kms

\*/

AbstractVehicleFactory autoFactory = FactoryProvider.getVehicleFactory("Auto")

Vehicle personalAuto = autoFactory.getVehicle("Personal")

personalAuto.book(distance)

# Abstract Factory Design Pattern

## Advantages

- offers loose coupling in the code
- supports open close principle since the code can be easily extended for supporting new classes
- enforces consistent creation of objects

## Disadvantages

- increased complexity