12__15-08-2022

Monday, 15 August 2022 8:03 PM

Differences between singleton and static classes.

→ static is a keyword and singleton is a design pattern.

→ singleton will give you an object whereas static class will provide static methods.

→ singleton can implement interfaces, inherit from the other classes and it align with the OOPs concepts, whereas same is not aligned with the static class.

→ singleton objects can be passed as a

reference whereas static objects cannot be passed as a reference to other methods or objects.

→ Singleton object can be passed as a reference, supports object disposal.

→ Singleton objects are stored on heap and they can also be cloned if required.

Real world usage of singleton

→ Information logging

→ Exception logging.

→ Connection pool management.

→ File management.

→ Device management such as printer spooling

→ App^n configuration management

→ Cache management

→ Session based shopping cart.

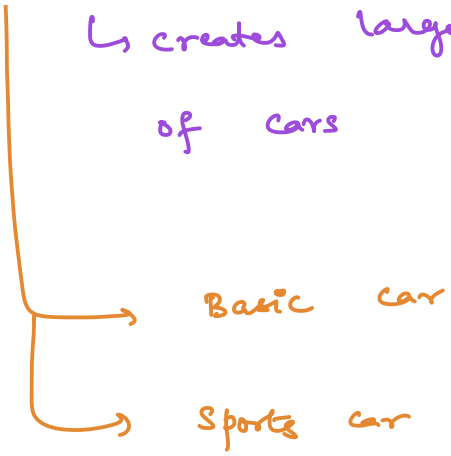
Factory design pattern

Factory → Mass production of similar kind of products

This design pattern provides support for mass production of similar kind of objects.

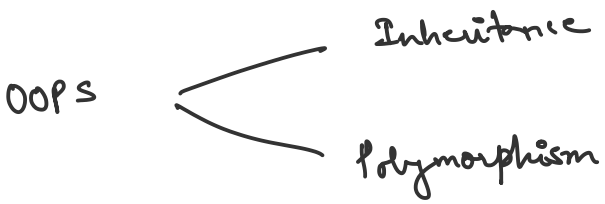
Car factory

↳ creates large number of cars



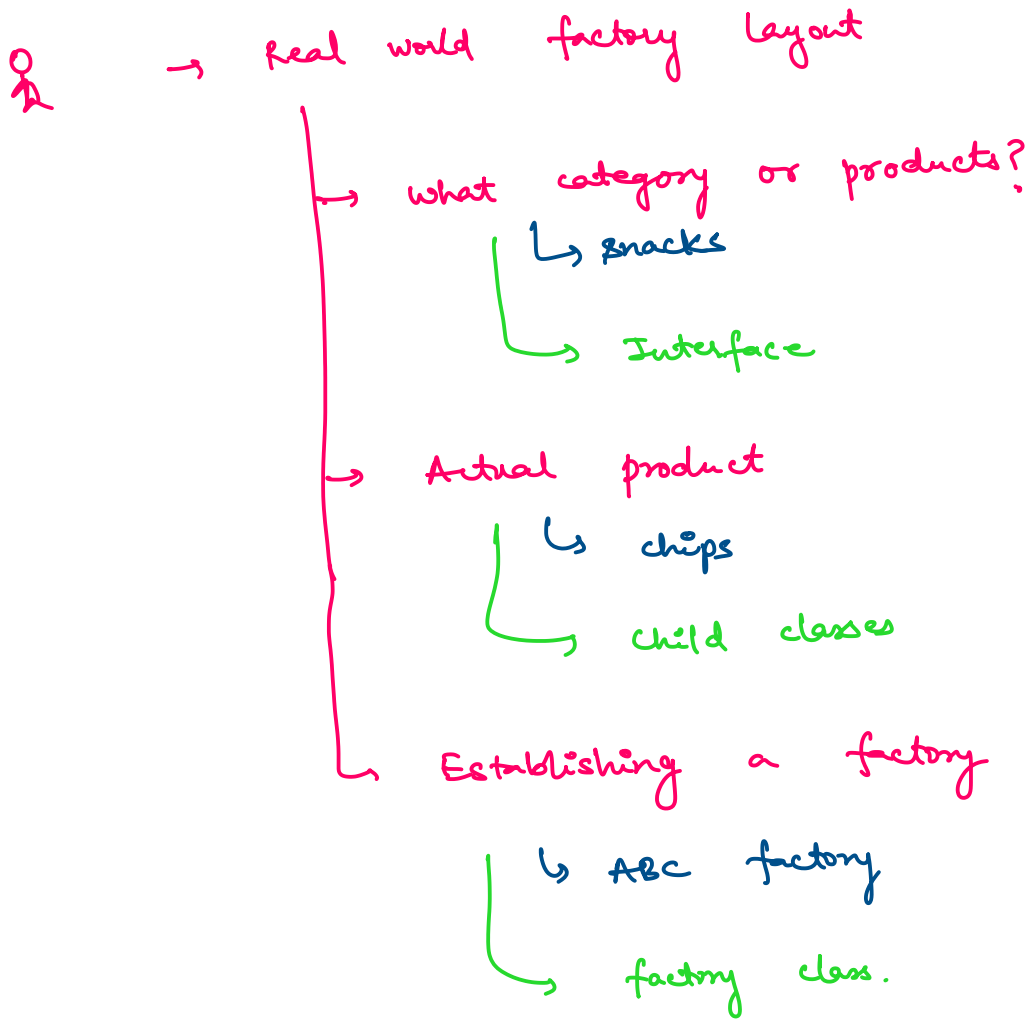
If you have to apply factory design pattern then there are certain things which you have to keep in your mind —

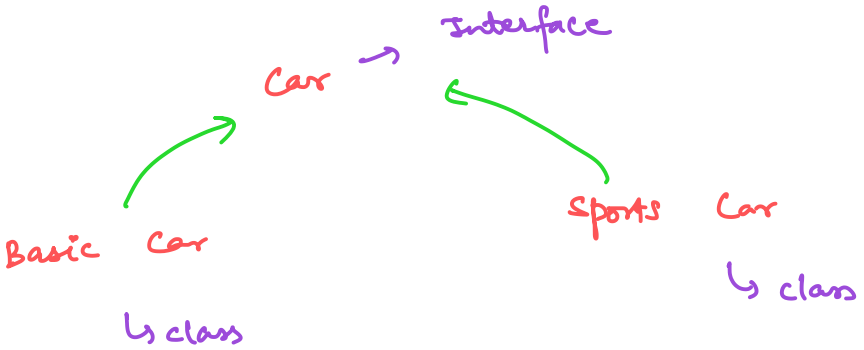
1. Use interface
   ↳ Along with this apply some OOPS concepts.



2. Create child classes.

3. Factory class.





Car factory (class)

↳ Responsible for creating different types of cars.

Gang of four definition

Define an interface for creating an object, but let sub-classes decide which class to instantiate.

The factory method lets a class defer instantiation it uses to sub-classes.

Factory pattern creates object without exposing the creation logic to the client and refer to newly created object using

a common interface.

Implementation guidelines

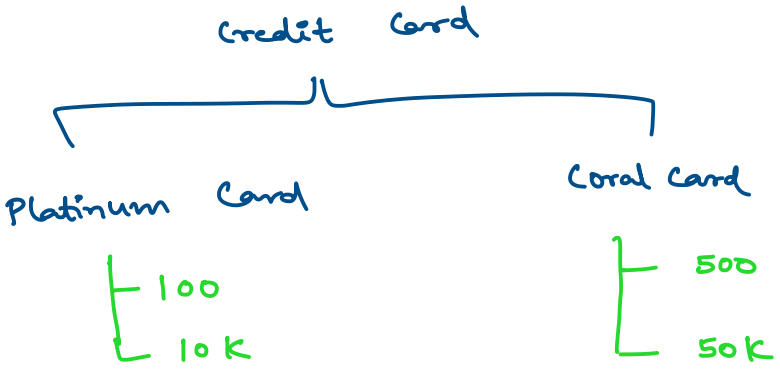
we need to choose factory pattern when—

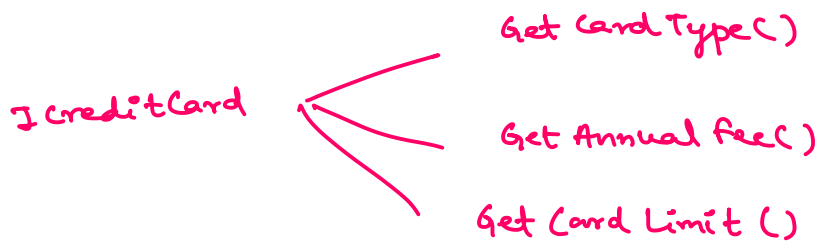
→ object needs to be extended to the subclasses.

→ The classes does not know what exact sub-classes it has to create.

→ The product implementation tend to change over time and the client remains unchanged.

Eg






```
public interface ICreditCard {
   public String GetCardType();
   public int GetAnnualFee();
   public int GetCardLimit();
}
```

```
public class PlatinumCard implements ICreditCard {
   @Override
   public String GetCardType() {
      return "Platinum Card";
   }

   @Override
   public int GetAnnualFee() {
      return 100;
   }

   @Override
   public int GetCardLimit() {
      return 10000;
   }
}
```

```
public class CoralCard implements ICreditCard {
   @Override
   public String GetCardType() {
      return "Coral Card";
   }
```

```
    @Override
    public int GetAnnualFee() {
        return 500;
    }

    @Override
    public int GetCardLimit() {
        return 50000;
    }
}
```

```
public class CreditCardFactory {
    public static ICreditCard GetCreditCardDetails(String cardType)
    {
        ICreditCard creditCard = null;

        if(cardType == "Platinum")
        {
            creditCard = new PlatinumCard();
        }
        else if(cardType == "Coral")
        {
            creditCard = new CoralCard();
        }
        else
        {
            System.out.println("Invalid card");
        }

        return creditCard;
    }
}
```

```
public class Program {
    public static void main(String[] args) {
        ICreditCard creditCard = CreditCardFactory.GetCreditCardDetails("Coral");

        if(creditCard != null)
        {
            System.out.println("Card type : " + creditCard.GetCardType());
            System.out.println("Annual fee : " + creditCard.GetAnnualFee());
```

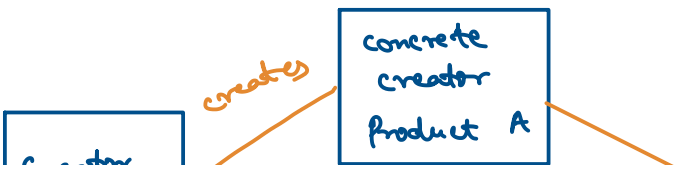
```
            System.out.println("Credit limit : " + creditCard.GetCardLimit());
        }
    }
}
```

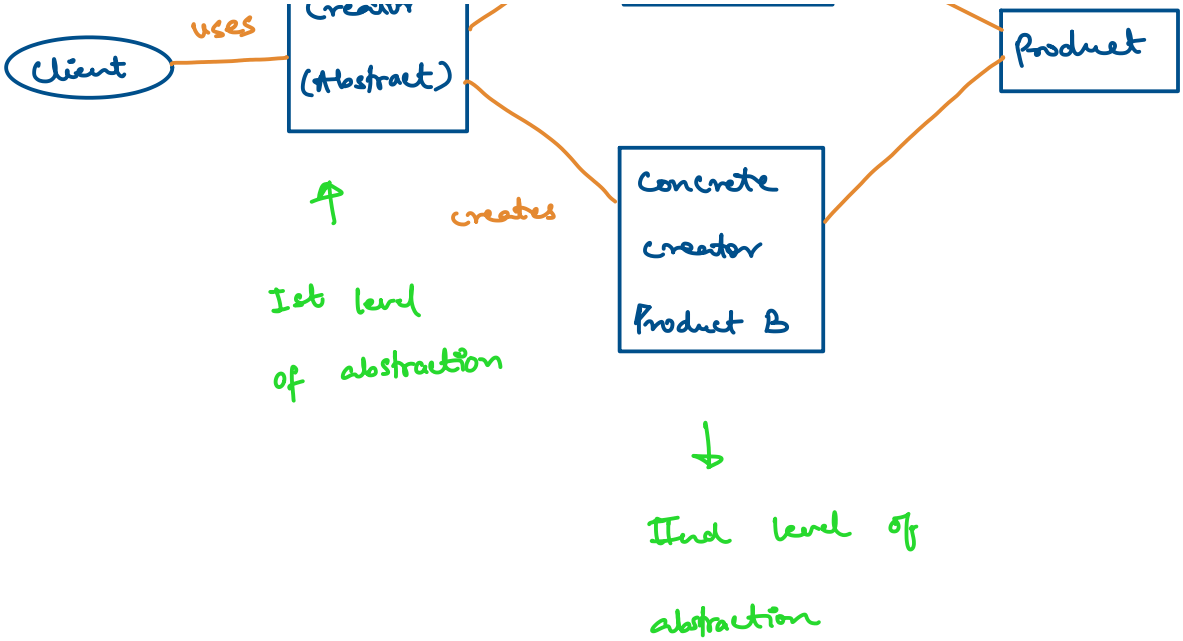
Output:
Card type : Coral Card
Annual fee : 500
Credit limit : 50000

Difference between simple factory and factory method pattern.

simple factory

Factory method pattern





Note –

The factory method pattern is also known as virtual constructor.

Advantages of factory design pattern

→ factory method pattern allows the sub- classes to choose the type of objects to create.

→ It promotes loose- coupling by eliminating the need to bind application specific

classes into the code. That means the code interacts solely with the resultant interface or abstract class, so that it will work with any classes that implement that interface or that extends that abstract class.

Implementation steps —

1. Create an interface.
2. Create concrete classes implementing the same interface.
3. Create a factory to generate objects based on given information.
4. Use the factory to get objects of concrete

class by passing information such as type.

```
public interface ICreditCard {
   public String GetCardType();
   public int GetAnnualFee();
   public int GetCardLimit();
}
```

```
public class PlatinumCard implements ICreditCard {
   @Override
   public String GetCardType() {
      return "Platinum Card";
   }

   @Override
   public int GetAnnualFee() {
      return 100;
   }

   @Override
   public int GetCardLimit() {
      return 10000;
   }
}
```

```
public class CoralCard implements ICreditCard {
   @Override
   public String GetCardType() {
      return "Coral Card";
   }

   @Override
```

```
    public int GetAnnualFee() {
        return 500;
    }

    @Override
    public int GetCardLimit() {
        return 50000;
    }
}
```

```
public abstract class CreditCardFactory {
    protected abstract ICreditCard CreateFactory();

    public ICreditCard CardFactory()
    {
        return CreateFactory();
    }
}
```

```
public class CoralFactory extends CreditCardFactory {
    @Override
    protected ICreditCard CreateFactory() {
        return new CoralCard();
    }
}
```

```
public class PlatinumFactory extends CreditCardFactory {
    @Override
    protected ICreditCard CreateFactory() {
        return new PlatinumCard();
    }
}
```

```
public class Program {
   public static void main(String[] args) {
      String cardType = "Platinum";

      ICreditCard creditCard = null;

      if(cardType == "Platinum")
      {
         creditCard = new PlatinumFactory().CardFactory();
      }
      else if(cardType == "Coral")
      {
         creditCard = new CoralFactory().CardFactory();
      }
      else
      {
         System.out.println("Invalid card");
      }


      System.out.println("Printing card details");
      if(creditCard != null)
      {
         System.out.println("Card type : " + creditCard.GetCardType());
         System.out.println("Annual fee : " + creditCard.GetAnnualFee());
         System.out.println("Credit limit : " + creditCard.GetCardLimit());
      }
   }
}
```

Output:

```
Printing card details
Card type : Platinum Card
Annual fee : 100
Credit limit : 10000
```