Factory Design Pattern

Factory method says to separate the object creation logic from the business logic.

It is three types:

1. Simple Factory
2. Factory Method
3. Abstract Factory Method

1. Simple Factory Design Principle

What is Simple factory?

It is a utility or class with a static ***method*** to ***create objects*** based on ***input*** (e.g., a ***string*** or ***enum***).

Let's understand the concept with an **example**.

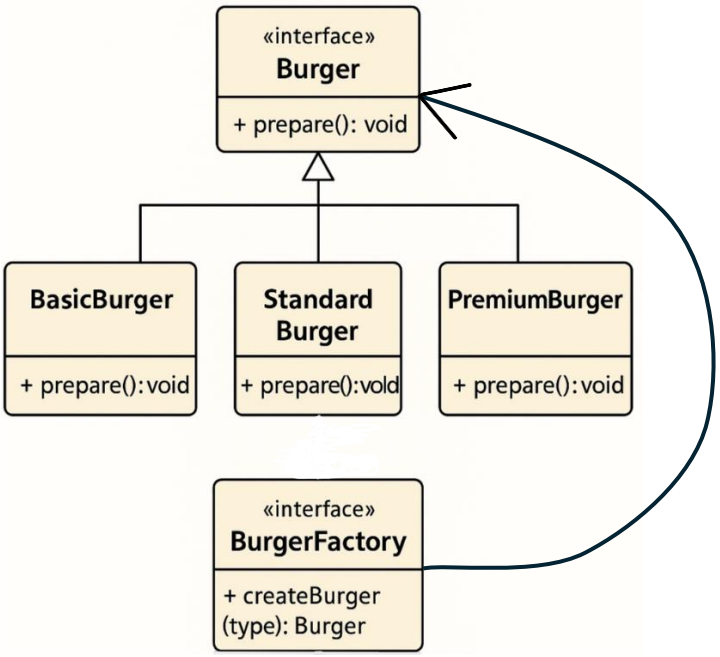
Suppose we have an ***interface*** named ***Burger*** 🍔 that declares a method ***prepare()***. There are three concrete implementations of this interface: ***BasicBurger***, ***StandardBurger***, and ***PremiumBurger***. Each of these classes overrides the ***prepare()*** method to define its ***specific preparation logic***.

In this context, Burger represents the ***product***, and we need a ***factory*** to ***create instances*** of different burger ***types***.

We have two options:

1. Instantiate the required Burger objects ***manually*** whenever needed.
2. Encapsulate the creation logic within a ***factory class*** that returns the appropriate Burger instance based on input.

To implement the second approach, we define a ***BurgerFactory*** interface with a method ***createBurger(String type)***. This method takes a burger type as a parameter and returns the corresponding Burger object.



Sample code:

```
public interface Burger {
   void prepare();
}

public class BasicBurger implements Burger {
   @Override
   public void prepare() {
      System.out.println("--------Preparing basic burger--------");
   }
}

public class StandardBurger implements Burger {
   @Override
   public void prepare() {
      System.out.println("--------Preparing standard burger--------");
   }
}

public class PremiumBurger implements Burger {
   @Override
   public void prepare() {
      System.out.println("--------Preparing premium burger--------");
   }
}
```

```
public class BurgerFactory {
   public Burger createBurger (String type) {
      type = type.toLowerCase();
      switch (type) {
         case "basic" -> {
            return new BasicBurger();
         }
         case "standard" -> {
            return new StandardBurger();
         }
         case "premium" -> {
            return new PremiumBurger();
         }
         default -> throw new RuntimeException("Burger not found");
      }
   }
}
```

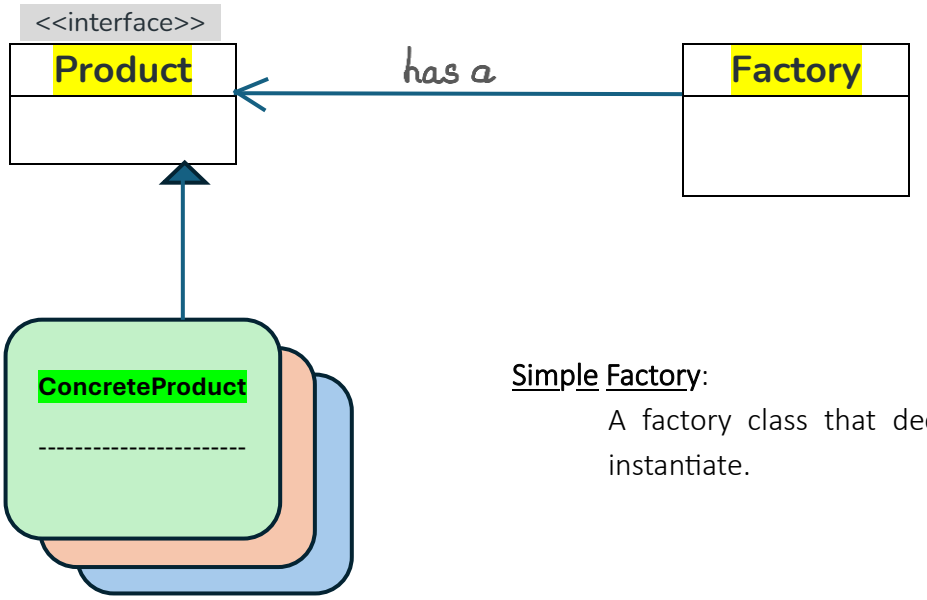
----------Client-code----------

```
public class Client {
   public static void main(String[] args) {
      BurgerFactory factory = new BurgerFactory();
      Burger burger = factory.createBurger("premium");
      burger.prepare();
   }
}
```

----------Client-code-output----------

```
--------Preparing premium burger--------
```

Standard UML of simple factory:

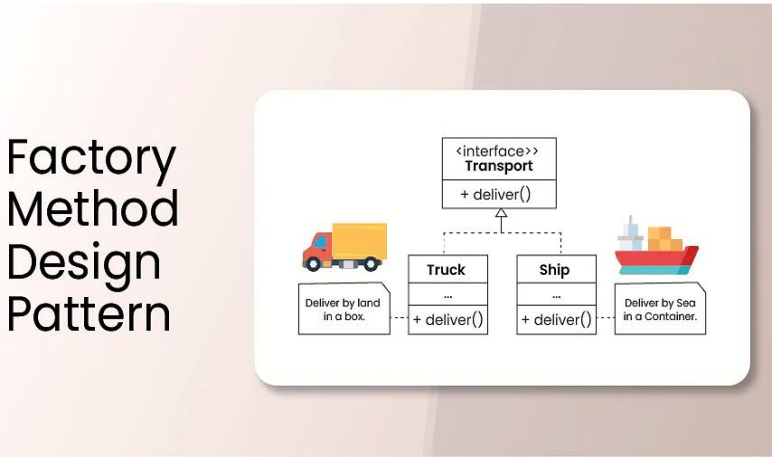


Simple Factory:

A factory class that decides which concrete class to be instantiate.

2. Factory Method Design Pattern

What is the Factory Method Design Pattern?



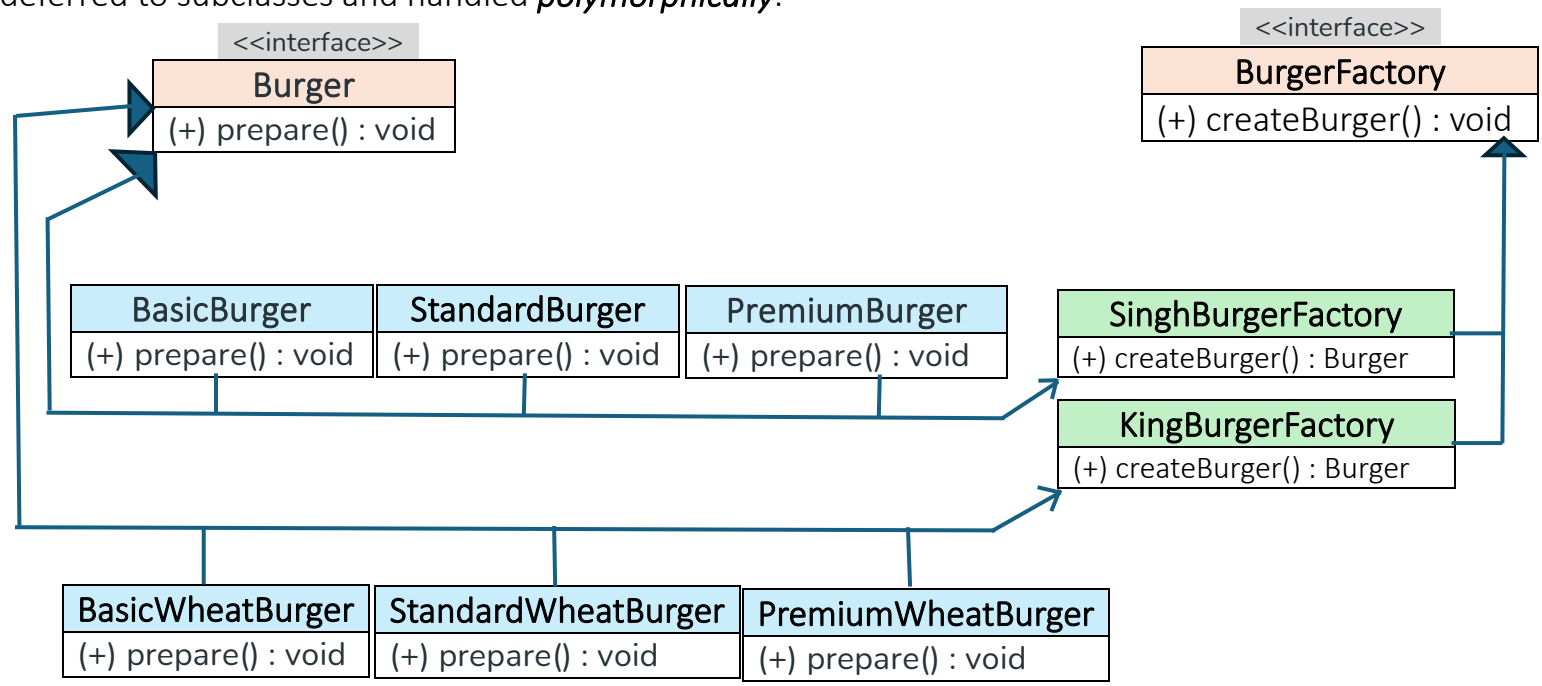
The Factory Method Design Pattern is a *creational design pattern* used in software development. It provides *an interface for creating objects* in a *superclass* while *allowing subclasses* to *specify the types* of *objects* they *create*.

- This pattern simplifies the *object creation process* by placing it in a *dedicated method*, promoting *loose coupling* between the *object creator* and the *objects* themselves.
- This approach enhances *flexibility*, *extensibility*, and *maintainability*, enabling *subclasses* to *implement* their *own factory methods* for *creating specific object types*.

Let's extend the previous example to better understand the *Factory Method Design Pattern*.
Initially, we had three types of *burgers* 🍔 : *BasicBurger*, *StandardBurger*, and *PremiumBurger*, all created using a **single factory** responsible for **preparing** these variants. Now, suppose we introduce **three** additional burger types that are healthier and made from wheat-based ingredients.
To handle this enhancement, we define *two separate factories*:

- *SinghBurgerFactory*, responsible for creating the regular burger variants.
- *KingBurgerFactory*, responsible for preparing the healthier wheat-based burgers.

We begin by defining a *BurgerFactory* **interface** that declares the method *createBurger*(*String type*). The two concrete factories, *SinghBurgerFactory* and *KingBurgerFactory*, implement this interface and provide their own logic to determine which specific Burger subclass to instantiate based on the **input type**.
This design **delegates** the **creation logic** to **subclasses**, allowing each factory to decide which burger to prepare—**demonstrating the core principle of the Factory Method pattern**, where the object creation is deferred to subclasses and handled *polymorphically*.



Product

```
public interface Burger {
   void prepare();
}
public class BasicBurger implements Burger {
   @Override
   public void prepare() {
      System.out.println("--preparing Basic Burger--");
   }
}
public class StandardBurger implements Burger {
   @Override
   public void prepare() {
      System.out.println("--preparing Standard Burger--");
   }
}
public class PremiumBurger implements Burger {
   @Override
   public void prepare() {
      System.out.println("--preparing Premium Burger--");
   }
}
public class BasicWheatBurger implements Burger {
   @Override
   public void prepare() {
      System.out.println("--preparing Basic Wheat Burger--");
   }
}
public class StandardWheatBurger implements Burger {
   @Override
   public void prepare() {
      System.out.println("--preparing Standard Wheat Burger--");
   }
}
public class PremiumWheatBurger implements Burger {
   @Override
   public void prepare() {
      System.out.println("--preparing Premium Wheat Burger--");
   }
}
```

Factory

```
public interface BurgerFactory {
   Burger createBurger(String type);
}
public class SinghBurgerFactory implements BurgerFactory {
   @Override
   public Burger createBurger(String type) {
      type = type.toLowerCase();
      switch (type) {
         case "basic" -> {
            return new BasicBurger();
         }
         case "standard" -> {
            return new StandardBurger();
         }
         case "premium" -> {
            return new PremiumBurger();
         }
         default ->
             throw new RuntimeException("Burger not found");
      }
   }
}
public class KingBurgerFactory implements BurgerFactory {
   @Override
   public Burger createBurger(String type) {
      type = type.toLowerCase();
      switch (type) {
         case "basic" -> {
            return new BasicWheatBurger();
         }
         case "standard" -> {
            return new StandardWheatBurger();
         }
         case "premium" -> {
            return new PremiumWheatBurger();
         }
         default ->
             throw new RuntimeException("Burger not found");
      }
   }
}
```

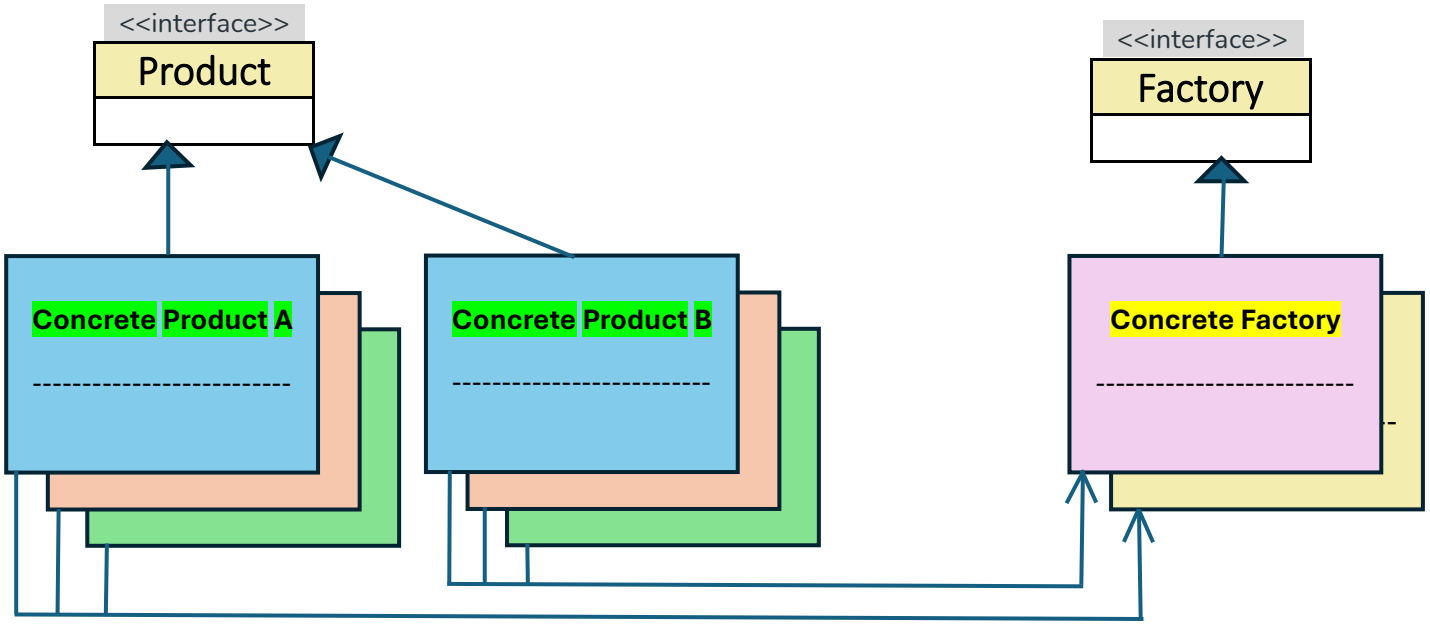
----------Client-Code----------

```
public class Client {
   public static void main(String[] args) {
      BurgerFactory kingFactory = new KingBurgerFactory();
      BurgerFactory singhFactory = new SinghBurgerFactory();
      Burger burger1 = kingFactory.createBurger("premium");
      Burger burger2 = singhFactory.createBurger("standard");
      burger1.prepare();
      burger2.prepare();
   }
}
```

----------Client-Code-Output----------

```
--preparing Premium Wheat Burger--
--preparing Standard Burger--
```

Standard UML of Factory Method design pattern:



Factory Method:
Define an interface for creating objects but allow sub-classes to decide which class to instantiate.

3. Abstract Factory Method Design Pattern

What is the Abstract Factory Method Design Pattern?

The *abstract factory method design pattern* provides an *interface* for creating *families* of *related* or *dependent* objects *without specifying* their *concrete classes*. It promotes *flexibility* and *modularity* by allowing you to switch between different families of objects without altering the client code that uses them.

- *Abstract Factory Method* is almost same as *Factory Method* and is considered as *another layer* of *abstraction over factory pattern*.
- Abstract Factory patterns work around *a super-factory* which creates *other factories*.
- At runtime, the *abstract factory* is coupled with any desired *concrete factory* which can *create objects* of the desired type.

Let's extend our ongoing burger 🍔 example to illustrate the Abstract Factory Design Pattern.
Previously, we had two concrete factories:

1. *SinghFactory* — responsible for creating regular burgers (BasicBurger, StandardBurger, PremiumBurger)
2. *KingFactory* — responsible for creating wheat-based burgers (BasicWheatBurger, StandardWheatBurger, PremiumWheatBurger)

Now, suppose the factory owners decide to expand their offerings to include **pizzas** 🍕 as well.

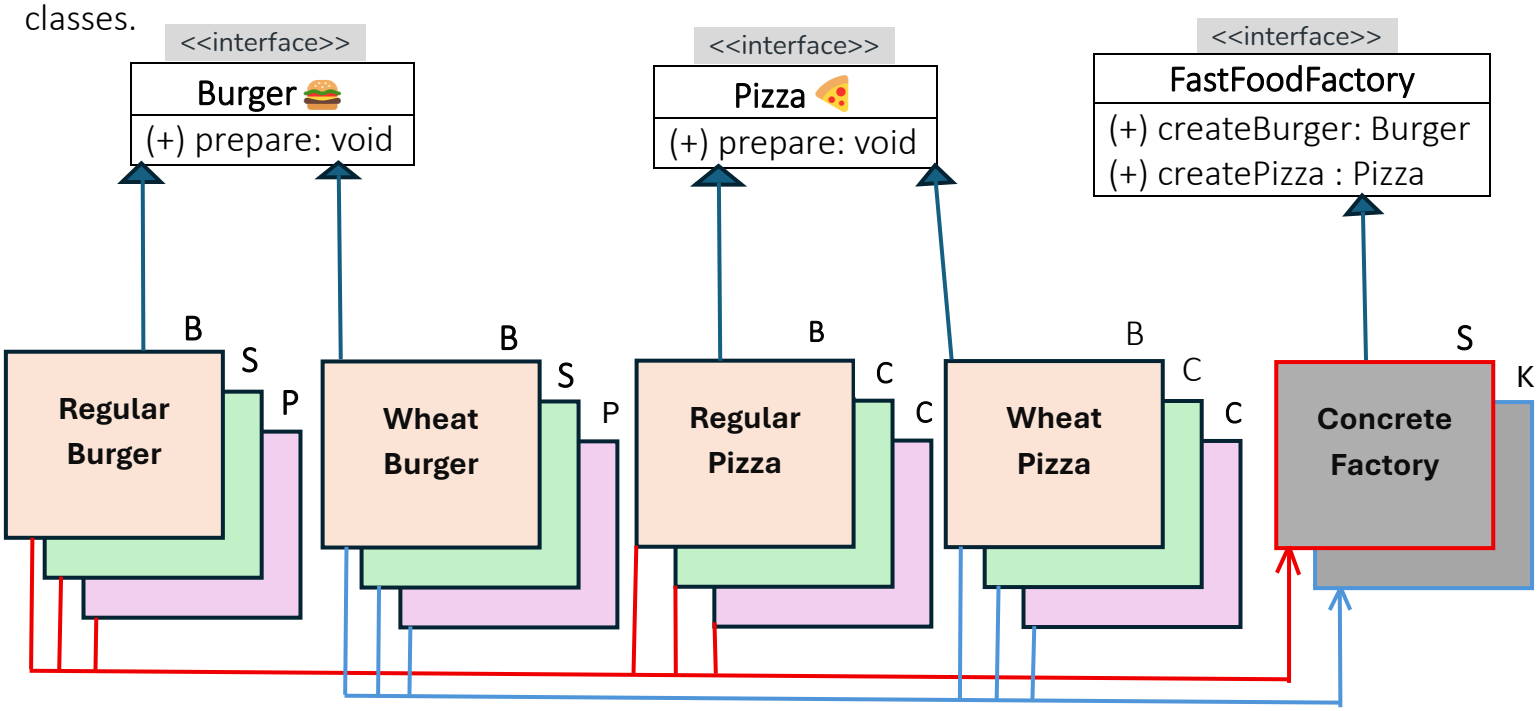
- ***SinghFactory*** will now produce **standard pizzas**: BasicPizza, CheesePizza, and ChickenPizza.
- ***KingFactory*** will offer **wheat-based pizzas**: BasicWheatPizza, CheeseWheatPizza, and ChickenWheatPizza.

To support this new structure, we define an ***abstract factory interface*** called ***FastFoodFactory***, which includes two methods:

- *createBurger*(String type) : Burger
- *createPizza*(String type) : Pizza

Each **concrete factory** (SinghFactory and KingFactory) implements this interface and provides the appropriate product variants depending on the brand (regular or wheat-based).

This approach demonstrates the essence of the **Abstract Factory Pattern**: Providing an interface for creating ***families of related products*** (in this case, **burgers** and **pizzas**) without specifying their concrete classes.



Sample Code:

----------Client-Code----------

```
public class Client {
    public static void main(String[] args) {
        String burgerType = "standard";
        String pizzaType = "chicken";

        FastFoodFactory factory1 = new SinghFastFoodFactory();
        FastFoodFactory factory2 = new KingFastFoodFactory();

        Burger burger = factory1.createBurger(burgerType);
        Pizza pizza = factory2.createPizza(pizzaType);

        burger.prepare();
        pizza.prepare();
    }
}
```

----------Client-Code-Output----------

```
🍔 Preparing Standard Burger 🍔
🍕 Preparing Chicken Wheat Pizza 🍕
```

Product

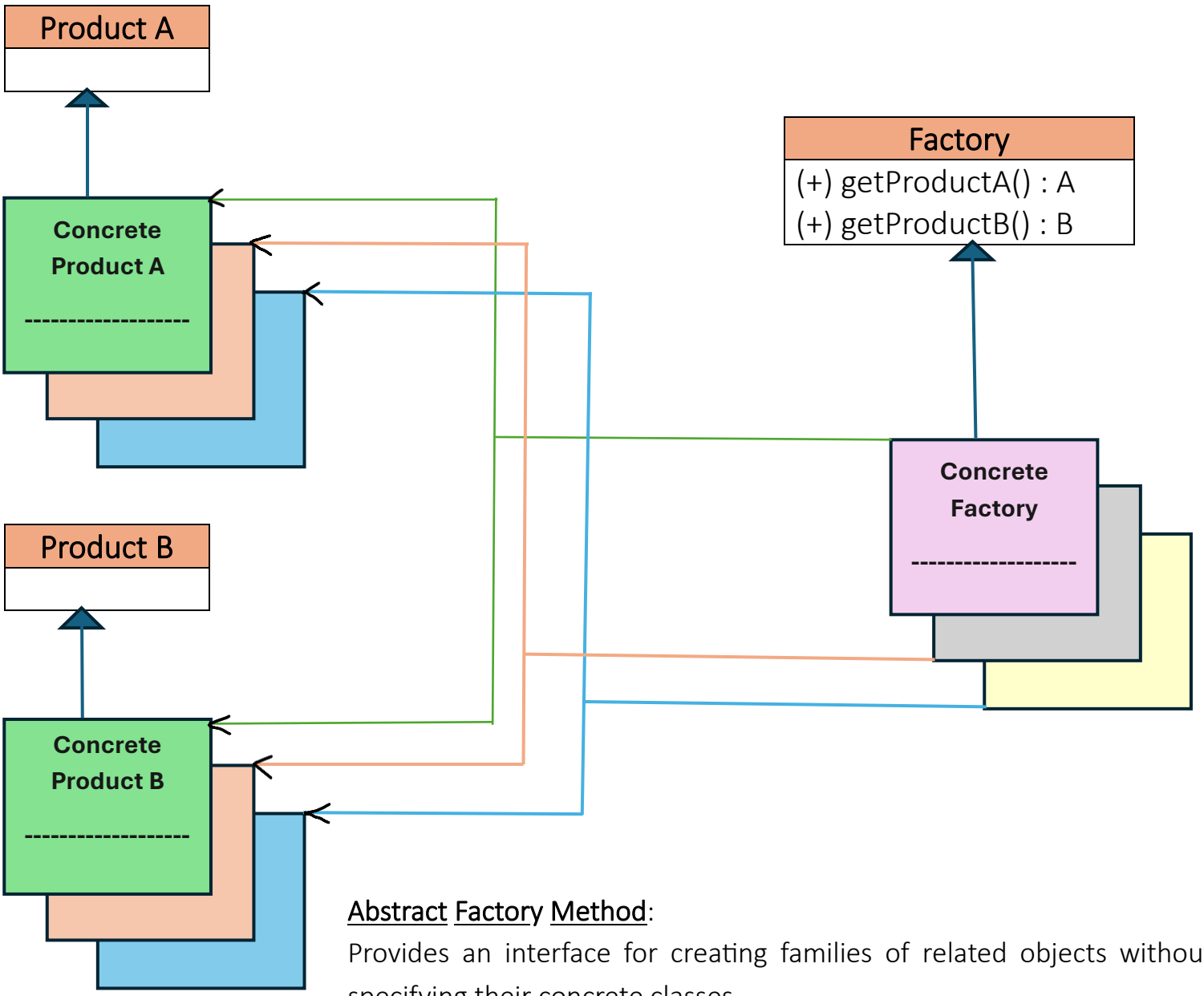
```
public interface Burger {
   void prepare();
}
public class BasicBurger implements Burger {
   @Override
   public void prepare() {
      System.out.println(" 🍔 Preparing Basic Burger 🍔 ");
   }
}
public class StandardBurger implements Burger {
   @Override
   public void prepare() {
      System.out.println(" 🍔 Preparing Standard Burger 🍔 ");
   }
}
public class PremiumBurger implements Burger {
   @Override
   public void prepare() {
      System.out.println(" 🍔 Preparing Premium Burger 🍔 ");
   }
}


public interface Pizza {
   void prepare();
}
public class BasicPizza implements Pizza {
   @Override
   public void prepare() {
      System.out.println(" 🍕 Preparing Basic Pizza 🍕 ");
   }
}
public class CheesePizza implements Pizza {
   @Override
   public void prepare() {
      System.out.println(" 🍕 Preparing Cheese Pizza 🍕 ");
   }
}
public class ChickenPizza implements Pizza {
   @Override
   public void prepare() {
      System.out.println(" 🍕 Preparing Chicken Pizza 🍕 ");
   }
}
public class BasicWheatPizza implements Pizza {
   @Override
   public void prepare() {
      System.out.println(" 🍕 Preparing Basic Wheat Pizza 🍕 ");
   }
}
public class CheeseWheatPizza implements Pizza {
   @Override
   public void prepare() {
      System.out.println(" 🍕 Preparing Cheese Wheat Pizza 🍕 ");
   }
}
public class ChickenWheatPizza implements Pizza {
   @Override
   public void prepare() {
      System.out.println(" 🍕 Preparing Chicken Wheat Pizza 🍕 ");
   }
}
```

Factory

```
public interface FastFoodFactory {
   Burger createBurger(String type);
   Pizza createPizza(String type);
}
public class SinghFastFoodFactory implements FastFoodFactory {
   @Override
   public Burger createBurger(String type) {
      type = type.toLowerCase();
      switch (type) {
         case "basic" -> { return new BasicBurger();}
         case "standard" -> { return new StandardBurger();}
         case "premium" -> {return new PremiumBurger();}
         default ->
               throw new RuntimeException("Burger not found!!");
      }
   }
   @Override
   public Pizza createPizza(String type) {
      type = type.toLowerCase();
      switch (type) {
         case "basic" -> {return new BasicPizza();}
         case "cheese" -> {return new CheesePizza();}
         case "chicken" -> {return new ChickenPizza();}
         default ->
               throw new RuntimeException("Pizza not found!!");
      }
   }
}
public class KingFastFoodFactory implements FastFoodFactory {
   @Override
   public Burger createBurger(String type) {
      type = type.toLowerCase();
      switch (type) {
         case "basic" -> {return new BasicWheatBurger();}
         case "standard" -> {return new StandardWheatBurger();}
         case "premium" -> {return new PrimumWheatBurger();}
         default ->
               throw new RuntimeException("Burger not found!!");
      }
   }
   @Override
   public Pizza createPizza(String type) {
      type = type.toLowerCase();
      switch (type) {
         case "basic" -> {return new BasicWheatPizza();}
         case "cheese" -> {return new CheeseWheatPizza();}
         case "chicken" -> {return new ChickenWheatPizza();}
         default ->
               throw new RuntimeException("Pizza not found!!");
      }
   }
}
```

@sibasundarj8

Standard UML of Abstract Factory Method design pattern:



Abstract Factory Method:

Provides an interface for creating families of related objects without specifying their concrete classes.