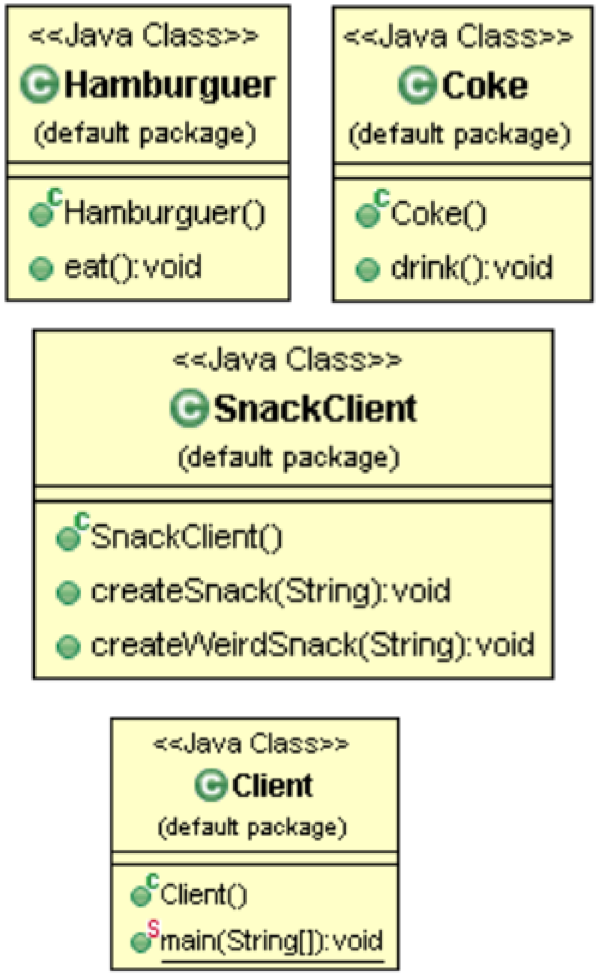
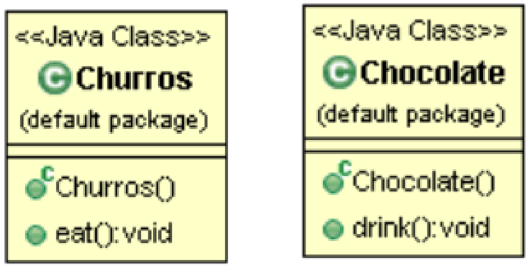
Design Patterns: Factory Method - Abstract Factory

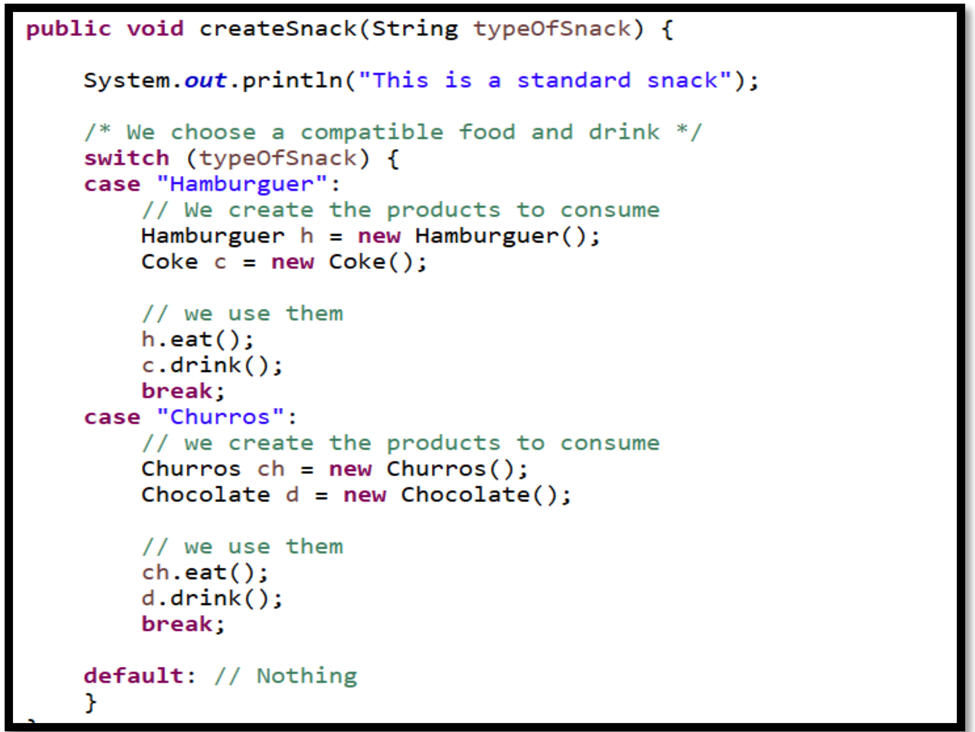
Example

Snack: food + drink

- Scenarios: Chocolate and Burger





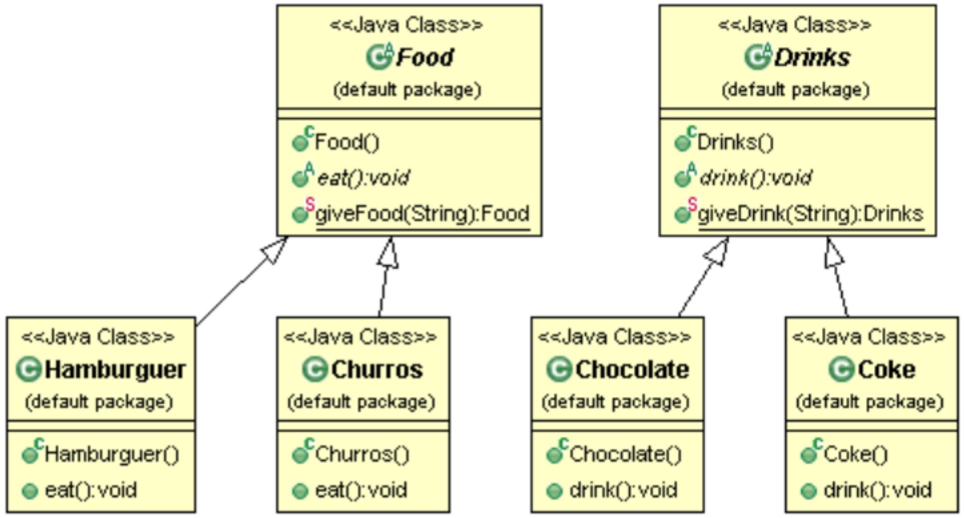


```java
public void createSnack(String typeOfSnack) {

    System.out.println("This is a standard snack");

    /* We choose a compatible food and drink */
    switch (typeOfSnack) {
    case "Hamburguer":
        // We create the products to consume
        Hamburguer h = new Hamburguer();
        Coke c = new Coke();

        // we use them
        h.eat();
        c.drink();
        break;
    case "Churros":
        // we create the products to consume
        Churros ch = new Churros();
        Chocolate d = new Chocolate();

        // we use them
        ch.eat();
        d.drink();
        break;

    default: // Nothing
    }
```

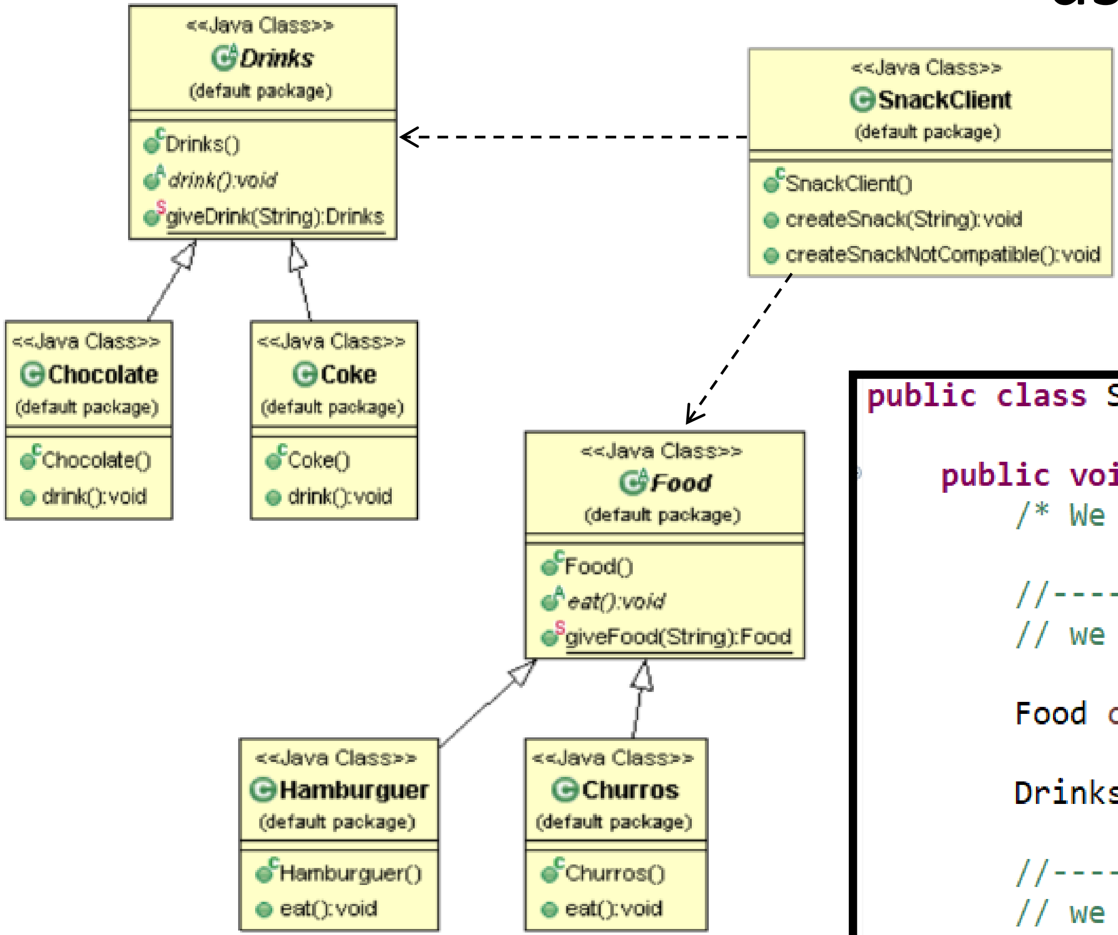
Factory Method: Separate the creation from the use of the “products”

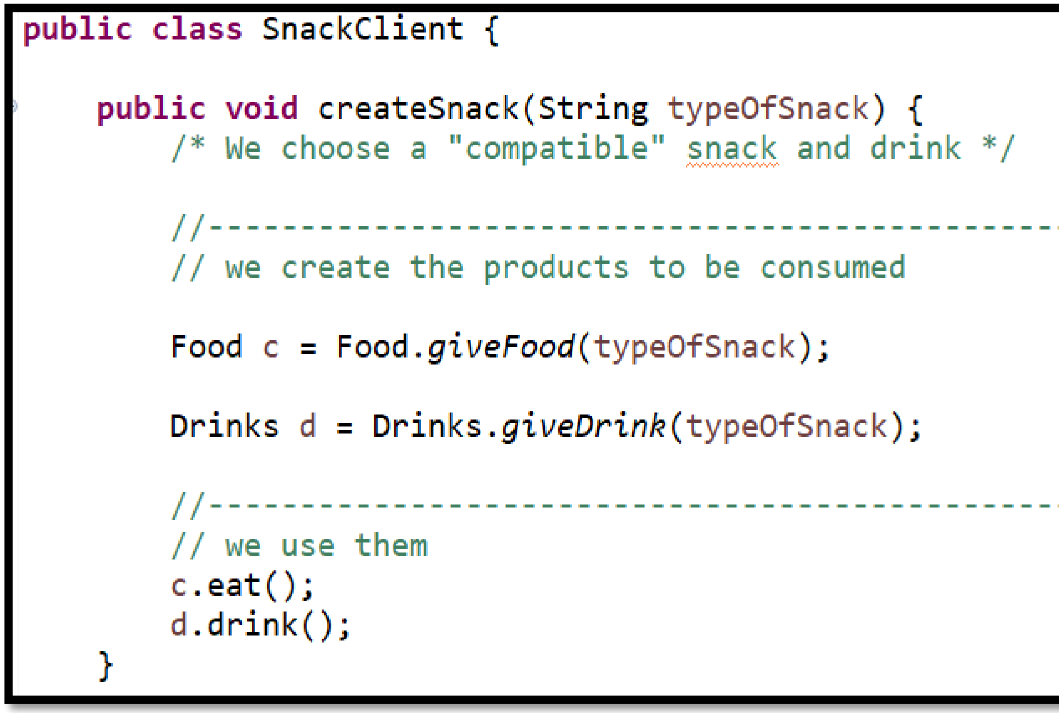
```
public abstract class Drinks {

    public abstract void drink();

    public static Drinks giveDrink(String type) {

        switch (type) {
        case "Hamburguer":
            return new Coke();

        case "Churros":
            return new Chocolate();

        /* ADD MORE DRINKS HERE */
        default:
            return null;
        }
    }
}
```

```
public abstract class Food {
    public abstract void eat();

    public static Food giveFood(String type) {
        switch (type) {
        case "Hamburguer":
            return new Hamburguer();

        case "Churros":
            return new Churros();

        /* ADD MORE FOOD HERE */
        default:
            return null;
        }
    }
}
```



Factory Method: Separate the creation from the use of the "products"

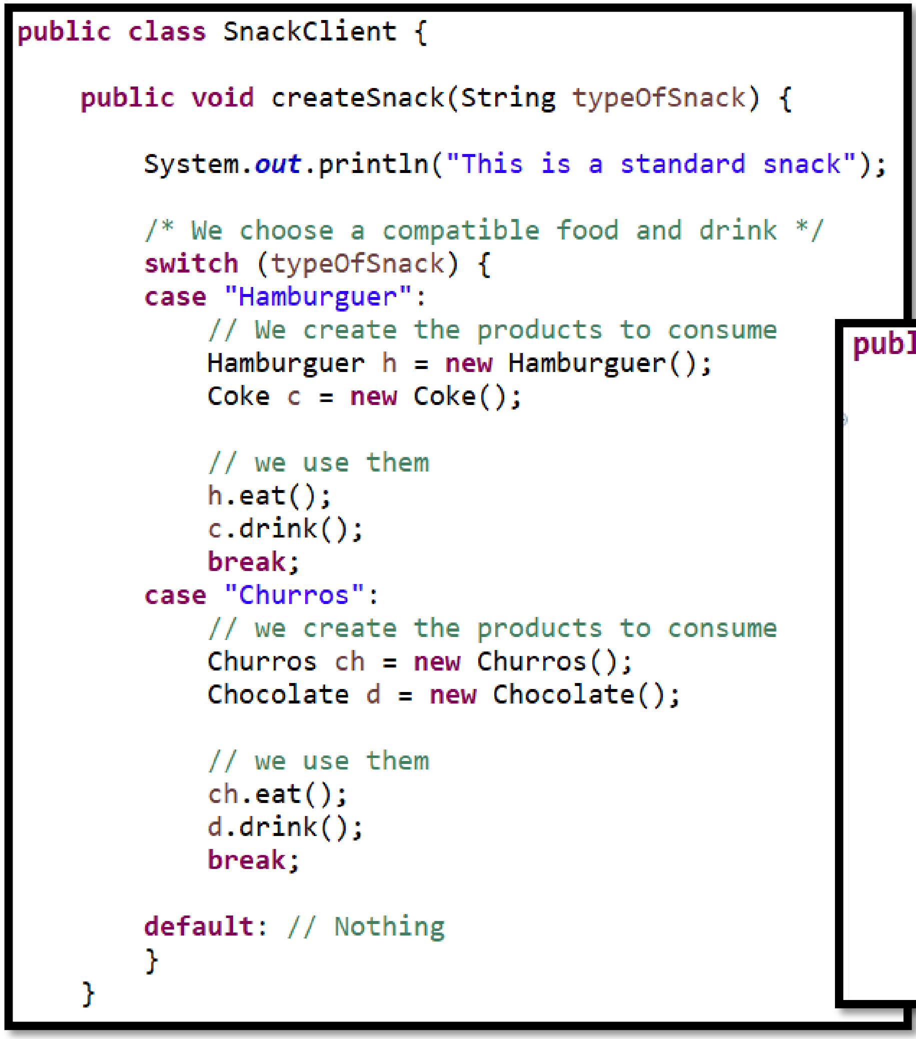




```
public class SnackClient {

    public void createSnack(String typeOfSnack) {
        /* We choose a "compatible" snack and drink */

        //-----------------------------------------
        // we create the products to be consumed

        Food c = Food.giveFood(typeOfSnack);

        Drinks d = Drinks.giveDrink(typeOfSnack);

        //-----------------------------------------
        // we use them
        c.eat();
        d.drink();
    }
```

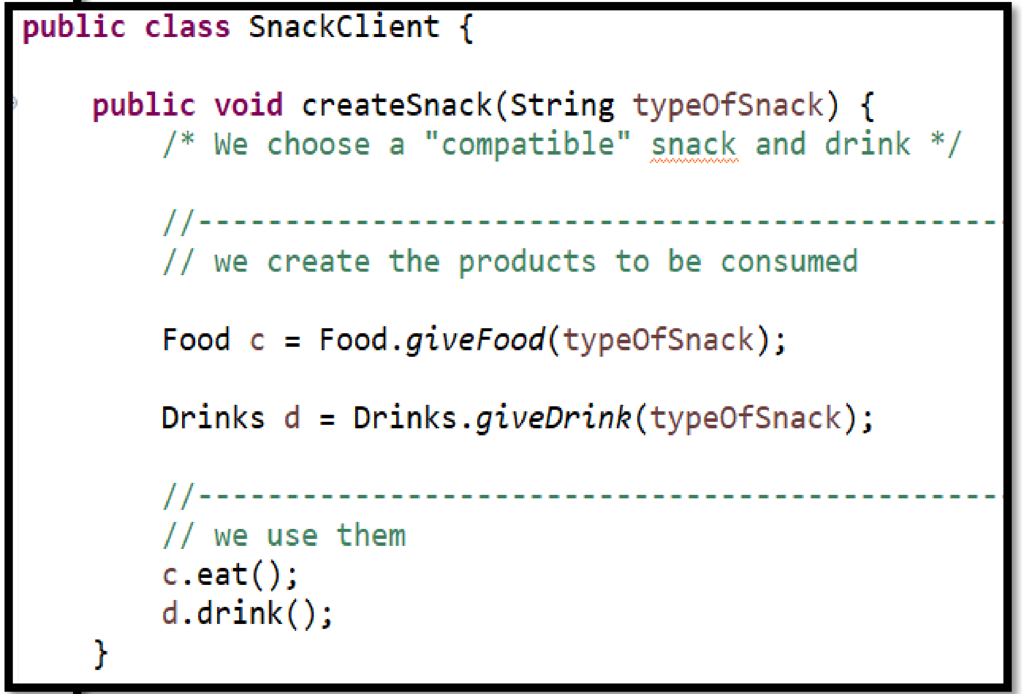
Factory Method: Separate the creation from the use of the “products”

Without pattern



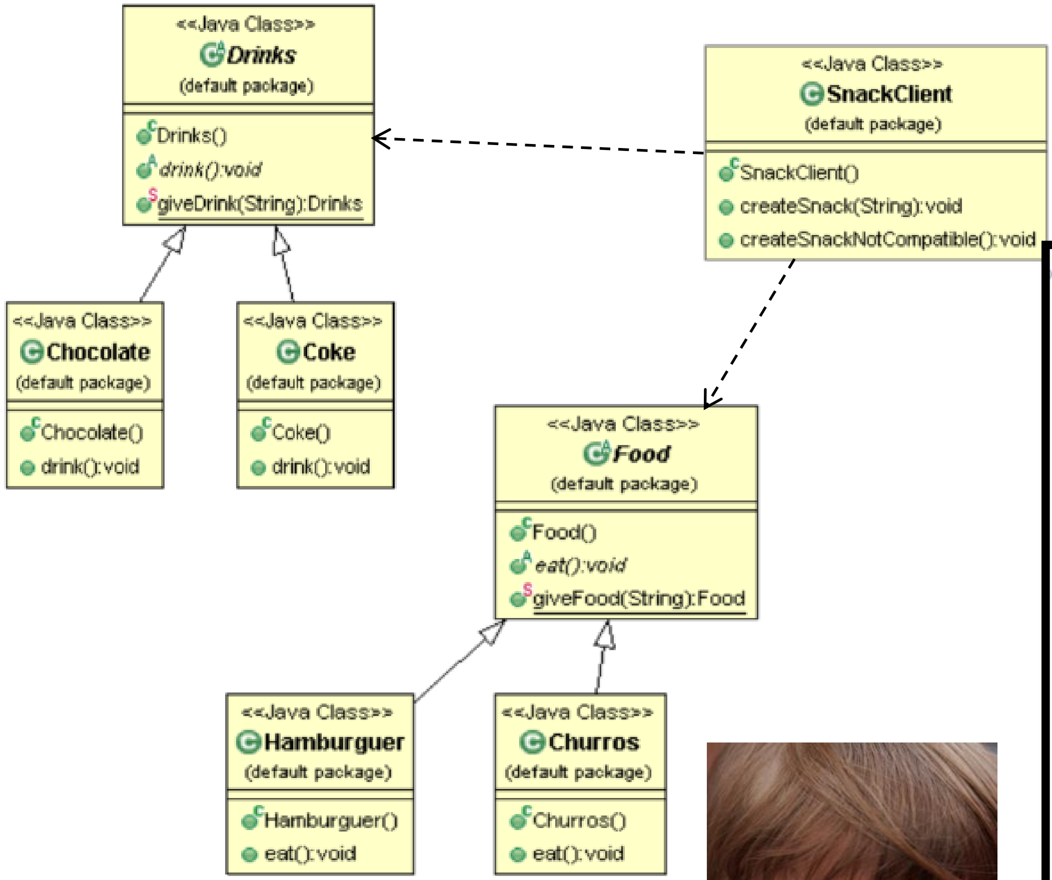
```
public class SnackClient {

    public void createSnack(String typeOfSnack) {

        System.out.println("This is a standard snack");

        /* We choose a compatible food and drink */
        switch (typeOfSnack) {
        case "Hamburguer":
            // We create the products to consume
            Hamburguer h = new Hamburguer();
            Coke c = new Coke();

            // we use them
            h.eat();
            c.drink();
            break;
        case "Churros":
            // we create the products to consume
            Churros ch = new Churros();
            Chocolate d = new Chocolate();

            // we use them
            ch.eat();
            d.drink();
            break;

        default: // Nothing
        }
    }
```

Factory Method(s)



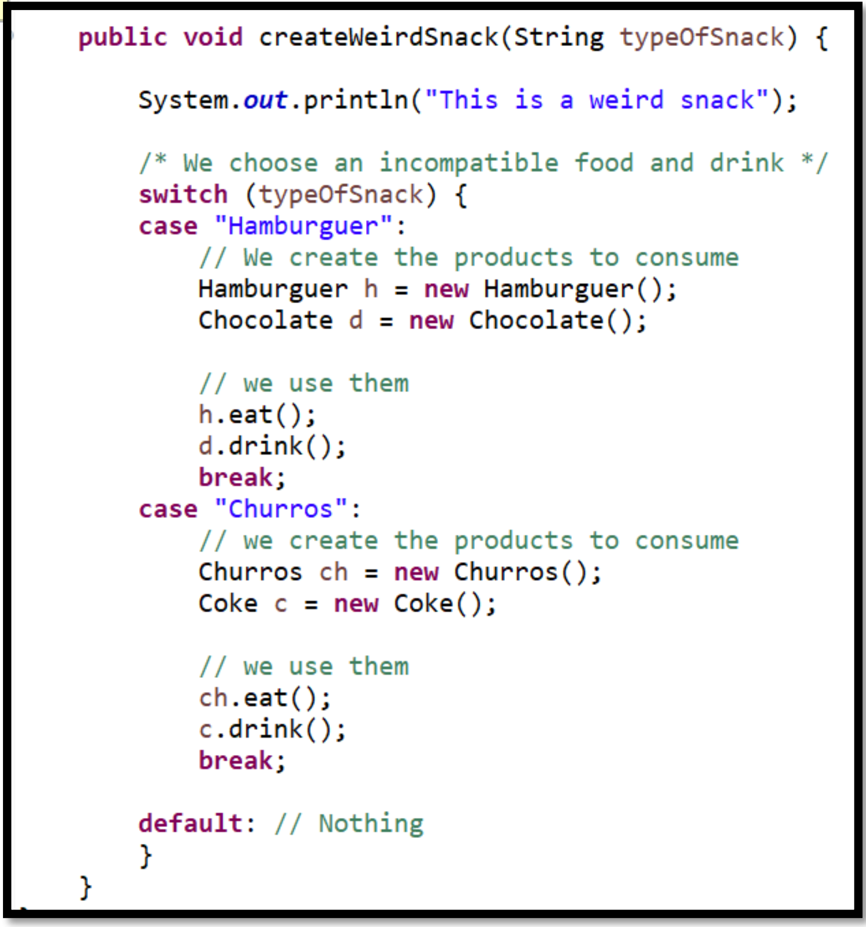
```
public class SnackClient {

    public void createSnack(String typeOfSnack) {
        /* We choose a "compatible" snack and drink */

        //-----------------------------------------
        // we create the products to be consumed

        Food c = Food.giveFood(typeOfSnack);

        Drinks d = Drinks.giveDrink(typeOfSnack);

        //-----------------------------------------
        // we use them
        c.eat();
        d.drink();
    }
```

Problem: compatible "products"?



Syntactically ✓

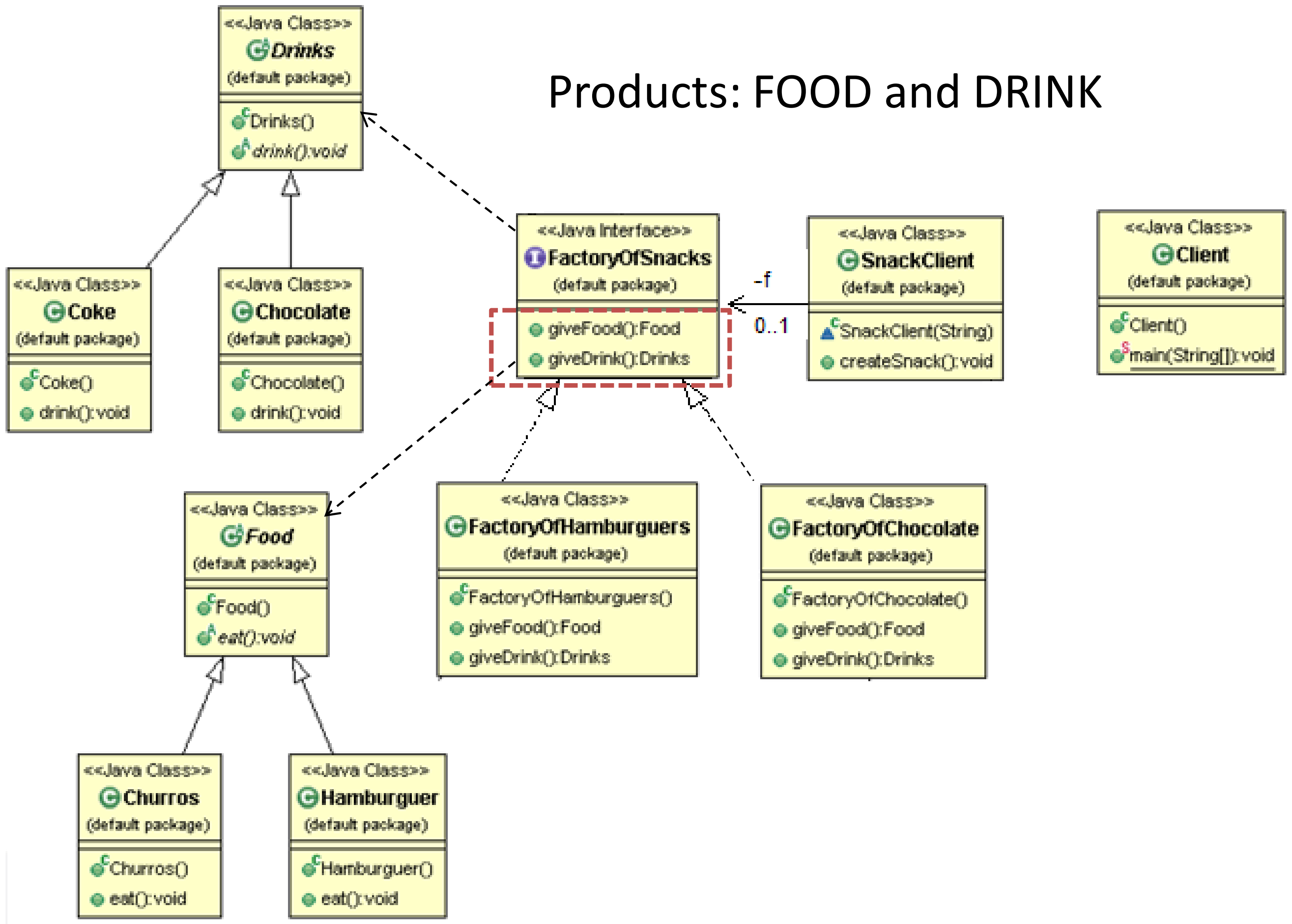
Semantically ✗



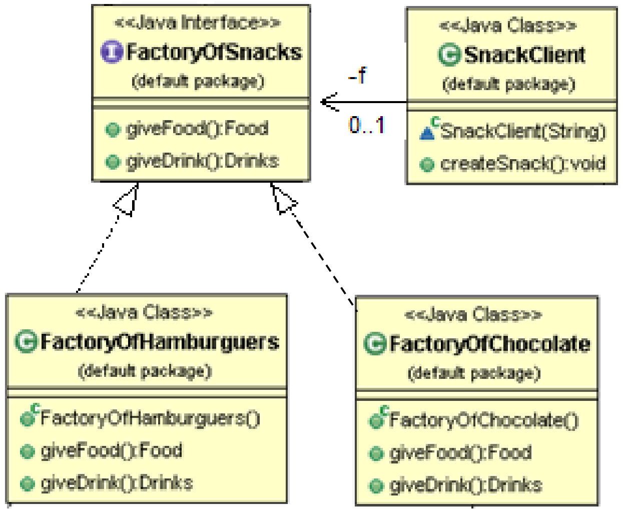
```
    public void createWeirdSnack(String typeOfSnack) {

        System.out.println("This is a weird snack");

        /* We choose an incompatible food and drink */
        switch (typeOfSnack) {
        case "Hamburguer":
            // We create the products to consume
            Hamburguer h = new Hamburguer();
            Chocolate d = new Chocolate();

            // we use them
            h.eat();
            d.drink();
            break;
        case "Churros":
            // we create the products to consume
            Churros ch = new Churros();
            Coke c = new Coke();

            // we use them
            ch.eat();
            c.drink();
            break;

        default: // Nothing
        }
    }
```

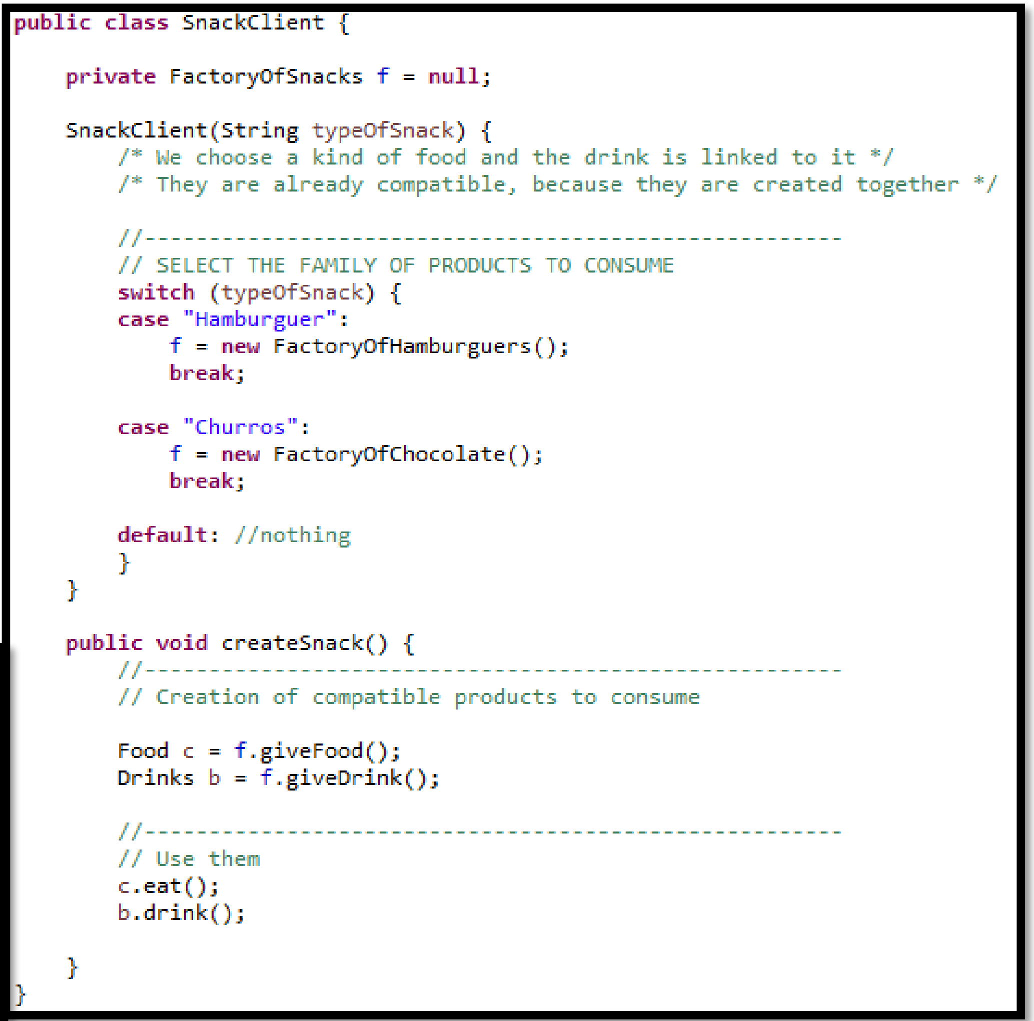
Solution: Abstract Factory

Products: FOOD and DRINK

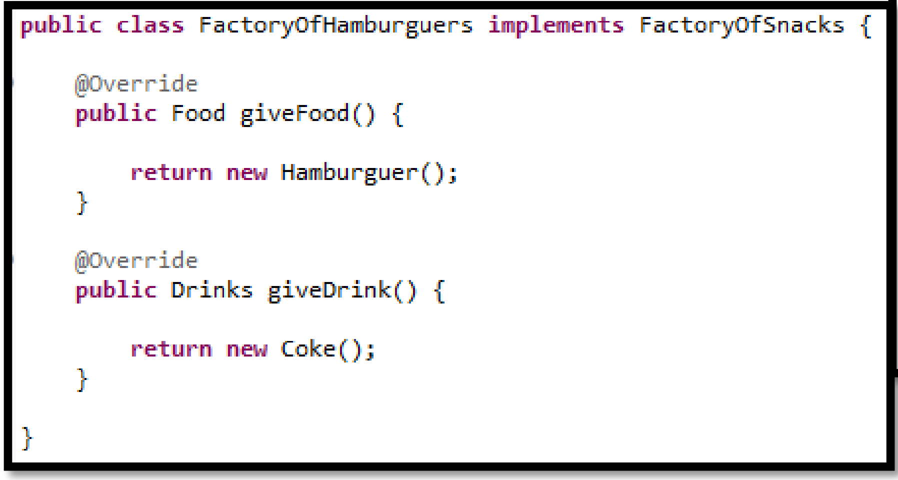


Solution: Abstract Factory





```java
public class SnackClient {

    private FactoryOfSnacks f = null;

    SnackClient(String typeOfSnack) {
        /* We choose a kind of food and the drink is linked to it */
        /* They are already compatible, because they are created together */

        //--------------------------------------------------------
        // SELECT THE FAMILY OF PRODUCTS TO CONSUME
        switch (typeOfSnack) {
        case "Hamburguer":
            f = new FactoryOfHamburguers();
            break;

        case "Churros":
            f = new FactoryOfChocolate();
            break;

        default: //nothing
        }
    }

    public void createSnack() {
        //--------------------------------------------------------
        // Creation of compatible products to consume

        Food c = f.giveFood();
        Drinks b = f.giveDrink();

        //--------------------------------------------------------
        // Use them
        c.eat();
        b.drink();

    }
}
```



```java
public class FactoryOfHamburguers implements FactoryOfSnacks {

    @Override
    public Food giveFood() {

        return new Hamburguer();
    }

    @Override
    public Drinks giveDrink() {

        return new Coke();
    }

}
```