Exercise #1 - S.O.L.I.D Principles Solutions

In this exercise let's go through all the SOLID steps and see how we can improve what is a bad design to start with into a much better design.

Question 1:

How does this code violate SRP? (Single Responsibility Principle)

```
class PizzaShop {
   constructor(name: string, city: string, zipCode: int){ }
   getName() { }
   changeAddress(city:string, zipCode: int) { }
}
```

Solution to question #1:

The main issue here is that the `PizzaShop` class is handling multiple responsibilities:

1. managing shop information (name, city, zipCode)
2. and also managing the address details of the shop.

This mixes the concerns of shop identity management with address management.

So, to adhere to SRP, we should separate concerns by creating different classes for different responsibilities.

```
// Handles shop details
class PizzaShop {
    constructor(name: string) { }
    getName() { }
}

// Handles address management
class Address {
    constructor(city: string, zipCode: int) { }
    changeAddress(city: string, zipCode: int) { }
}
```

Question 2:

How does this code violate OCP? (Open-Closed Principle)

```
class PizzaShop {
    constructor(name: string, address: Address){ }
    getName() { }
    getAddress() { }
}
class InvoiceService {
    generateInvoice(shop: MusicShop): String{
        let invoice = "";
        if(company instanceOf A)
          invoice = "format of invoice for A";
        if(company instanceOf B)
          invoice = "format of invoice for B";
        if(company instanceOf C)
          invoice = "format of invoice for C";
        return invoice;
    }
}
```

Solution to question #2:

The main issue is that the `InvoiceService` class directly checks the instance of the shop to generate invoices, leading to modifications in the `InvoiceService` class every time a new shop type is added. This mixes the concerns of shop identity management with address management.

So, to adhere to OCP, we will use polymorphism where each shop can have its own implementation of generating an invoice, avoiding modifications to the `InvoiceService` class when adding new shop types.

```
interface Shop {
    generateInvoice(): string;
}

class A implements Shop {
    generateInvoice(): string {
        return "format of invoice for A";
    }
}

class B implements Shop {
    generateInvoice(): string {
        return "format of invoice for B";
    }
}

class InvoiceService {
    generateInvoice(shop: Shop): string {
        return shop.generateInvoice();
    }
}
```

Question 3:

How does this code violate LCP? (Liskov Substitution Principle)

```
class PizzaShop {
    homeDelivery();
    //...
}
class A extends PizzaShop {
    homeDelivery() {
        return "delivery is free for all our customers";
    }
}
class B extends PizzaShop {
    takeaway() {
        throw new Exception('We do not have home delivery
service');
    }
}
```

Solution to question #3:

The main issue is that Class `B` changes the behavior by not supporting `homeDelivery` and instead introducing `takeaway`, which is not expected behavior for a subclass of `PizzaShop`.

To solve this issue we need to ensure that all subclasses of `PizzaShop` can be used interchangeably without altering the expected behavior. If `homeDelivery` is a common operation, all subclasses must support it.

```
class A extends PizzaShop {
    homeDelivery() {
        return "delivery is free for all our customers";
    }
}

// Ensure B either supports homeDelivery or is not a subclass
// of PizzaShop if it fundamentally differs.
class B extends PizzaShop {
    homeDelivery() {
        return "Home delivery is not available";
    }
}
```

Question 4:

How does this code violate ISP? (Interface Segregation Principle)

```
abstract class IPizzaShop{
    //traditional pizzerias
    getOvenBakedPizza()();
    getClassicalBakedPizza();
    //new wave pizzerias
    getElectricOvenBakedPizza();
    getPizzaPocketSquareBakedPizza();
    // all pizzerias
    getDrinks();
}
class TraditionalPizzeria implements IPizzaShop {
    getOvenBakedPizza() {
        //...          }
    getClassicalBakedPizza() {
        //...             }
    getDrinks() {
        //...   }
    getElectricOvenBakedPizza() {
        throw new Exception('We don't do that');
    }
    getPizzaPocketSquareBakedPizza() {
        throw new Exception('We don't do that');
    }
}
class NewWavePizzeria implements IPizzaShop {
    getElectricOvenBakedPizza() {
        //...                   }
    getPizzaPocketSquareBakedPizza() {
        //...                          }
    getDrinks() {
        //...   }
    getOvenBakedPizza() {
        throw new Exception('We don't do that');
    }
    getClassicalBakedPizza() {
        throw new Exception('We don't do that');
    }
}
```

Solution to question #4:

In this case the problem is that the `IPizzaShop` interface forces subclasses to implement methods that they do not need, like `getElectricOvenBakedPizza` for traditional pizzerias and `getOvenBakedPizza` for new wave pizzerias.

To solve this issue we will split the `IPizzaShop` interface into smaller, more specific interfaces so that implementing classes only need to adhere to the interfaces that are relevant to them.

```
interface TraditionalPizzaShop {
    getOvenBakedPizza();
    getClassicalBakedPizza();
    getDrinks();
}

interface NewWavePizzaShop {
    getElectricOvenBakedPizza();
    getPizzaPocketSquareBakedPizza();
    getDrinks();
}
```

Question 5:

How does this code violate the Dependency Inversion Principle?

```
class PizzaShop {
    getPayment() {
    }
    deliverPizza() {
    }
}
class Customer {
    makePayment() {
    }
    receivePizza() {
    }
}
class Delivery {
    constructor(customer: Customer, pizzaShop: PizzaShop) { }
    deliver() {
        customer.makePayment
        pizzaShop.getPayment
        pizzaShop.deliverPizza
        customer.receivePizza
    }
}
```

Solution to question #5:

The problem here is that the `Delivery` class is directly dependent on concrete classes `Customer` and `PizzaShop`, making it difficult to extend or modify the behavior of `Delivery` without changing these concrete classes.

We need to ensure that we depend on abstractions rather than concrete classes. To solve this we introduce interfaces or abstract classes that `Customer` and `PizzaShop` can implement.

```
interface IPaymentReceiver {
    getPayment();
}

interface IDeliveryReceiver {
    deliverPizza();
}

interface IPaymentMaker {
    makePayment();
}

interface IPizzaReceiver {
    receivePizza();
}

class Delivery {
    constructor(customer: IPaymentMaker & IPizzaReceiver,
pizzaShop: IPaymentReceiver & IDeliveryReceiver) { }
    deliver() {
        customer.makePayment();
        pizzaShop.getPayment();
        pizzaShop.deliverPizza();
        customer.receivePizza();
    }
}
```