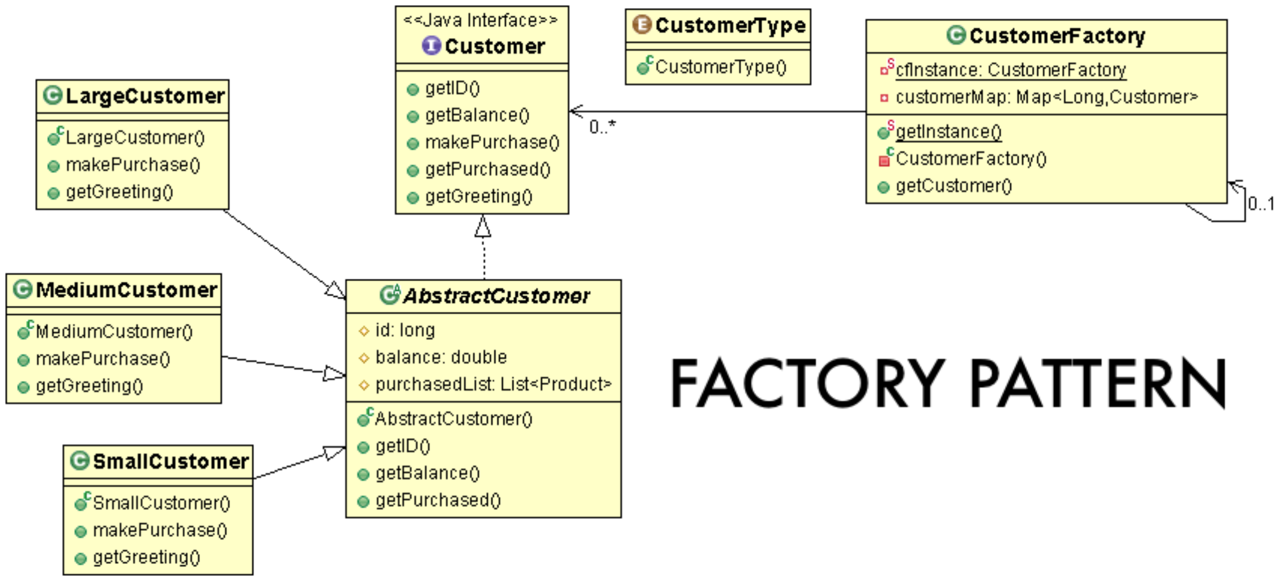
Software Design Pattern: The Factory Pattern

This repository is a demo for a Design Pattern known as the Factory Pattern.
The Factory Pattern allows one of several chosen types of concrete classs to be created that share a common interface, while hiding the creation logic from the user.

You can download all the java code for this demo in the eclipse directory format, compile it, and run the CustomerDemo class to see some output.

As seen in the diagram below, there are a number of classes required to implement the **Factory Pattern**.

In this demo, we have three different concrete implementations of a type of Customer that can be created.



The CustomerFactory is called by the client and requests a Customer by using a specific long id and a CustomerType.
The CustomerFactory first checks to see if a Customer with this id has already been created, and if not, then it creates one of three concrete implementations of Customer ... SmallCustomer, MediumCustomer, and LargeCustomer.

The three concrete XXXCustomer classes share a number of methods, which are defined in AbsractCustomer, which in turn implements the Customer Interface.

The goal of this Customer heirarchy is to separate diffrent types of customers, and this becomes evident if you look inside of any of the three concrete XXXCustomer classes and note the makePurchase() method has a slightly different restriction. The result being, the "larger" a Customer type you are, the larger your one-time purchases can be.

Creating Customers looks like this in the actual code of the client, where each Customer is created with a CustomerType, a type long that indicates the unique id, and an initial monetary double balance that Customer can use to buy Products ...

```
CustomerFactory customerFactory = CustomerFactory.getInstance();
```

```
Customer customerA = customerFactory.getCustomer(CustomerType.SMALL, 1459292, 25_000.00d);
Customer customerB = customerFactory.getCustomer(CustomerType.MEDIUM, 2245945, 60_000.00d);
Customer customerC = customerFactory.getCustomer(CustomerType.LARGE, 324772, 210_000.00d);
```

Another thing to note is that the CustomerFactory is also a Singleton, which means there can only be one CustomerFactory for the entire application. This is why CustomerFactory.getinstance() is called as shown above. We only have one instance of CustomerFactory and it is created in a Lazy Susan manner.
The CustomerFactory can keep track of how many Customers are created and store them in a Map that is checked each time a new Customer is created. We don't want to have two Customers with the same id!

Also in this demo is a class called Product, which has fields corresponding to a type String name and type double cost.
Now, the client creates as many Product(s) as desired as shown below ...

```
// create some dummy Product Objects and put them in a List
List productList = new ArrayList<>();
for( int i = 1; i <= 21; i++ ) {
        try {
                Product p = new Product("Product" + i, i*1_000d);
                productList.add(p);
        } catch( ProductCreationException ex ) {
                System.out.println( ex.getMessage() );
        }
}
```

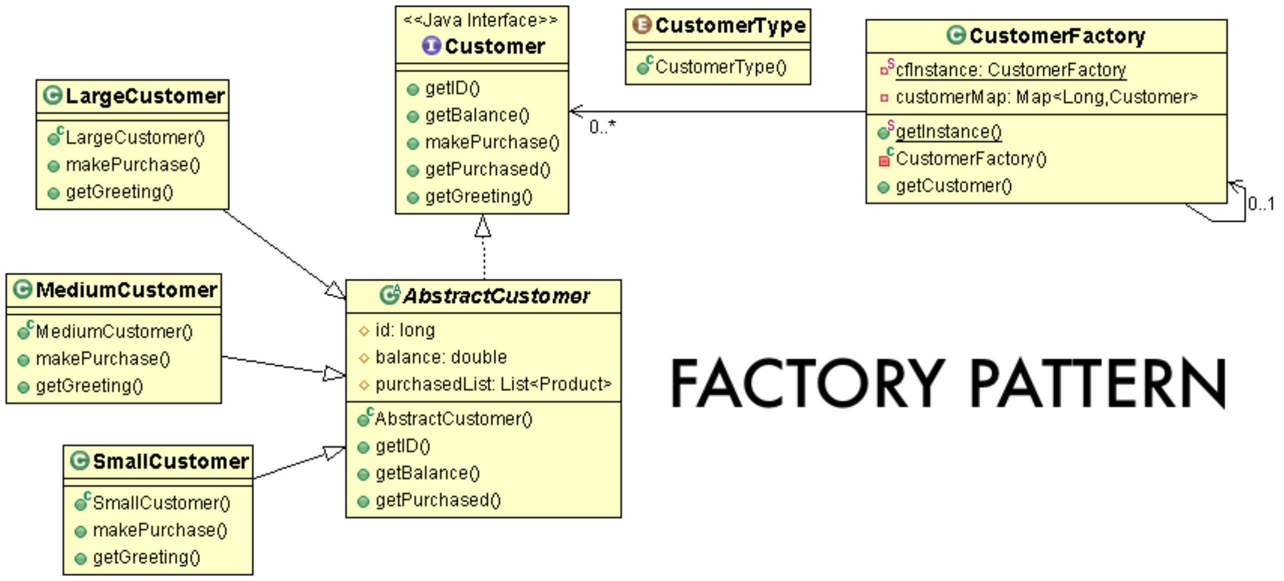
Then the client can have any of the Customer objects try and purchase any or all of the Product(s). Here we loop through all of the Product(s) in the List productList and have customerA, who is a SmallCustomer, attempt to buy them all ...

```
for( Product product: productList) {
        // customerA
        try {
                customerA.makePurchase(product);
        } catch( PurchaseAmountExceededException paec) {

        } catch( BadPurchaseException bpe ) {

        }
}
```

As suspected, the Customer's balance may reach 0 before all of the Products are purchased and hence any problem making a purchase causes an Exception to be thrown in the makePurchase method and we catch it with customized Exceptions called PurchaseAmountExceededException and BadPurchaseException. # Software-Design-Factory-Pattern This repository is a demo for a Design Pattern known as the Factory Pattern.
The Factory Pattern allows one of several chosen types of concrete classs to be created that share a common interface, while hiding the creation logic from the user.
You can download all the java code for this demo, compile it, and run the CustomerDemo class to see some output.
Then you can tweak any of the classes as you desire.

As we can see from the diagram below, there are a number of classes required to immplement this Pattern.

In this demo, we have three different concrete implementations of a type of Customer that can be created.



The CustomerFactory is called by the client and requests a Customer by using a specific long id and a CustomerType.
The CustomerFactory first checks to see if a Customer with this id has already been created, and if not, then it creates one of three concrete implementations of Customer ... SmallCustomer, MediumCustomer, and LargeCustomer.

The three concrete XXXCustomer classes share a number of methods, which are defined in AbsractCustomer, which in turn implements the Customer Interface.

The goal of this Customer heirarchy is to separate diffrent types of customers, and this becomes evident if you look inside of any of the three concrete XXXCustomer classes and note the makePurchase() method has a slightly different restriction. The result being, the "larger" a Customer type you are, the larger your one-time purchases can be.

Creating Customers looks like this in the actual code of the client, where each Customer is created with a CustomerType, a type long that indicates the unique id, and an initial monetary double balance that Customer can use to buy Products ...

```
CustomerFactory customerFactory = CustomerFactory.getInstance();
Customer customerA = customerFactory.getCustomer(CustomerType.SMALL, 1459292, 25_000.00d);
Customer customerB = customerFactory.getCustomer(CustomerType.MEDIUM, 2245945, 60_000.00d);
Customer customerC = customerFactory.getCustomer(CustomerType.LARGE, 324772, 210_000.00d);
```

Another thing to note is that the CustomerFactory is also a Singleton, which means there can only be one CustomerFactory for the entire application. This is why CustomerFactory.getinstance() is called as shown above. We only have one instance of CustomerFactory and it is created in a Lazy Susan manner.
The CustomerFactory can keep track of how many Customers are created and store them in a Map that is checked each time a new Customer is created. We don't want to have two Customers with the same id!

Also in this demo is a class called Product, which has fields corresponding to a type String name and type double cost.
Now, the client creates as many Product(s) as desired as shown below ...

```
// create some dummy Product Objects and put them in a List
List productList = new ArrayList<>();
for( int i = 1; i <= 21; i++ ) {
        try {
                Product p = new Product("Product" + i, i*1_000d);
                productList.add(p);
        } catch( ProductCreationException ex ) {
                System.out.println( ex.getMessage() );
        }
}
```

Then the client can have any of the Customer objects try and purchase any or all of the Product(s). Here we loop through all of the Product(s) in the List productList and have customerA, who is a SmallCustomer, attempt to buy them all ...

```
for( Product product: productList) {
        // customerA
        try {
                customerA.makePurchase(product);
        } catch( PurchaseAmountExceededException paec) {

        } catch( BadPurchaseException bpe ) {

        }
}
```

As suspected, the Customer's balance may reach 0 before all of the Products are purchased and hence any problem making a purchase causes an Exception to be thrown in the makePurchase method and we catch it with customized Exceptions called PurchaseAmountExceededException and BadPurchaseException.