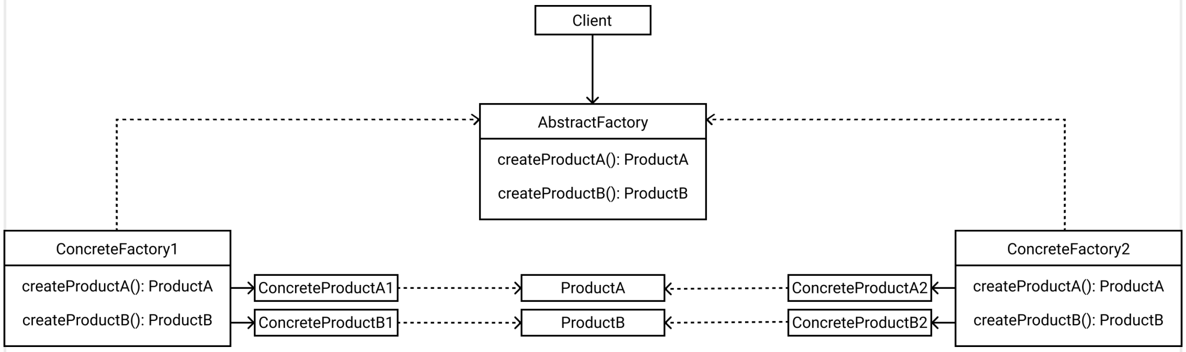
Abstract factory

Imagine that you are the boss of a factory, any factory producing any product you could imagine.
You have a perfect creational algorithm that is being used.
However, one day you decide to expand your business and open one more factory in another city.
That means that you have to encapsulate your creational algorithm for old and new factories. This is what the *Abstract Factory* is about.

The Abstract Factory pattern

Abstract Factory is one of the creational patterns. It encapsulates the creational process of related or dependent objects. The most interesting thing here is that there is no need to specify the concrete classes of these objects.


Client
AbstractFactory
createProductA(): ProductA
createProductB(): ProductB
ConcreteFactory1
createProductA(): ProductA
createProductB(): ProductB
ConcreteProductA1
ConcreteProductB1
ProductA
ProductB
ConcreteProductA2
ConcreteProductB2
ConcreteFactory2
createProductA(): ProductA
createProductB(): ProductB

Let's get down to business and create a *TableFactory*. We want to be an outstanding company, so like IKEA, we will be producing something that is easy to build. For this reason, it is also necessary to create the *ToolsFactory*. Before we begin implementing, it is important to clarify the definition of *Abstract*

Factory. This pattern is creational, so in general, it should not modify the provided objects, it should just provide them in a simple manner.

Example: TableFactory

First of all, let's define our *Table* interface and *Kitchen* and *Office* implementations of it:

```
interface Table {
   String getTable();
}

public class KitchenTable implements Table {
   @Override
   public String getTable() {
      return "This is a kitchen table";
   }
}

public class OfficeTable implements Table {
   @Override
   public String getTable() {
      return  "This is an office table";
   }
}
```

Secondly, we need tools which we will provide for the full kit:

```
interface TableTools {
   String getTool();
}

public class KitchenTableTool implements TableTools {
   @Override
   public String getTool() {
      return "These are kitchen table tools";
   }
}

public class OfficeTableTool implements TableTools {
   @Override
   public String getTool() {
      return "These are office table tools";
   }
}
```

Third things third, it is time to create an *AbstractFactory*. We have all the pieces we need for our relational objects. For example, *KitchenTableTools* are related to *KitchenTable*. That means that we need to encapsulate the creational process in a *TableFactory* and we will finally get the **Abstract Factory** pattern.

```
interface TableFactory {
   Table makeTable();
   TableTools makeTableTools();
}

public class KitchenTableFactory implements TableFactory {
   @Override
   public Table makeTable() {
      return new KitchenTable();
   }

   @Override
   public TableTools makeTableTools() {
      return new KitchenTableTool();
   }
}

public class OfficeTableFactory implements TableFactory {
   @Override
   public Table makeTable() {
      return new OfficeTable();
   }

   @Override
   public TableTools makeTableTools() {
      return new OfficeTableTool();
   }
}
```

Finally, let's test what we have created. Two lucky customers come to our shop, a *cook* and an *office manager*. They want to buy tables for work.

```
Table table;
TableTools tableTools;

TableFactory kitchenTableFactory = new KitchenTableFactory();
TableFactory officeTableFactory = new OfficeTableFactory();

System.out.println("-I work as a cook. I need a kitchen table");
System.out.println("-Got It! Give me a sec,- Calling the KitchenTableFactory. -
Bring me the KitchenTable with KitchenTableTools");
Thread.sleep(5000);
```

```
table = kitchenTableFactory.makeTable();
tableTools = kitchenTableFactory.makeTableTools();

System.out.println(table.getTable() + "\n" + tableTools.getTool());
System.out.println("-There they are!\n");
Thread.sleep(5000);

System.out.println("-I am an office manager. I need an office table");
System.out.println("-Got It! Give me a sec,- Calling the OfficeTableFactory. -
Bring me the OfficeTable with OfficeTableTools");
Thread.sleep(5000);

table = officeTableFactory.makeTable();
tableTools = officeTableFactory.makeTableTools();

System.out.println(table.getTable() + "\n" + tableTools.getTool());
System.out.println("-There they are!");
```

And the final output is:

```
-I work as a cook. I need a kitchen table
-Got It! Give me a sec,- Calling the KitchenTableFactory. - Bring me the
KitchenTable with KitchenTableTools
This is a kitchen table
These are kitchen table tools
-There they are!

-I am an office manager. I need an office table
-Got It! Give me a sec,- Calling the OfficeTableFactory. - Bring me the
OfficeTable with OfficeTableTools
This is an office table
These are office table tools
-There they are!
```

Conclusion

We've implemented a *TableFactory* that controls the creational process. Each factory implementation encapsulated its product. For example, *KitchenTableFactory* encapsulated *KitchenTable* and *KitchenTableTools*. One of the benefits is that we can be sure that we will get the correct *TableTools* with the corresponding *Table*.

1) Abstract Factory provides an interface (or abstract class) for creating a family of related objects without specifying their concrete classes.
2) Abstract Factory is based on composition: object creation occurs in a method that is accessed through the factory interface (or abstract class).
3) Abstract Factory ensures low coupling by decreasing concrete dependencies of classes in the running code.

Disadvantages of the pattern
What is the disadvantage of abstract factory?

abstract factory interface fixes the set of products that can be created

The statement: "abstract factory interface fixes the set of products that can be created", means this:
Eg: In the theory we had an abstract factory, TableFactory. And that TableFactory had fixed that it will only create 2 related products viz- Table and TableTools.

Now, say we identified that there is another related product that we'd like TableFactory to create (maybe - TableCloth). Now, this task isn't flexible at all. And frankly, kind of a pain in a lot of areas ;). Because we'd have to "change" the TableFactory interface (or abstract class). But the story doesn't end here. We'd have to do the exact same thing in all the concrete implementations (like - KitchenTableFactory, OfficeTableFactory) of this abstract factory (i.e. TableFactory). We should rather aim for minimal changes in the code, but look at the kind of mess this has created.