Design Patterns:

Types:

This is a Java Design Patterns project.

1.Creational

1.1.Singleton

1.2.Factory

1.3.AbstractFactory

1.4.Prototype

2.Structural

2.1.Adapter

2.2.Composite

2.3.Facade

2.4.Decorator

2.5.Proxy

3.Behavioural

3.1.Observer

3.2.Strategy

**Creational Design Patterns:**

1. Singleton
2. Factory
3. Abstract Factory
4. Builder
5. Prototype
6. **Singleton**

Singletons are used to create one and only one instance of a class in JVM.

They are used for logging, caching etc.

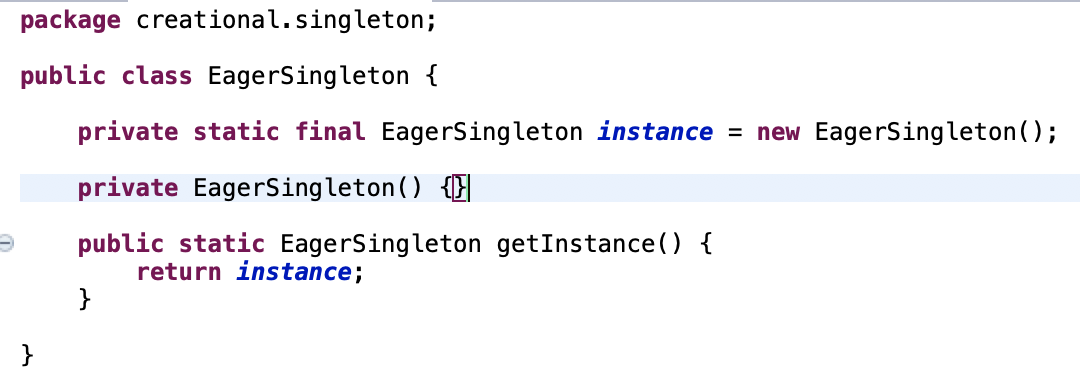
There are various ways to achieve singleton.

The underlying approach is to make the constructor private and expose a private instance of the class through a public static method which acts as a global access point.

**EagerSingleton**

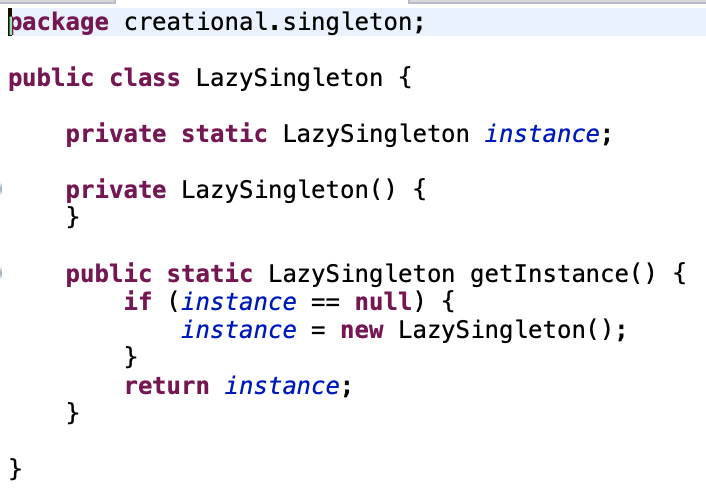
Here the singleton is created irrespective of whether anyone is going to use it

And it is marked as final.



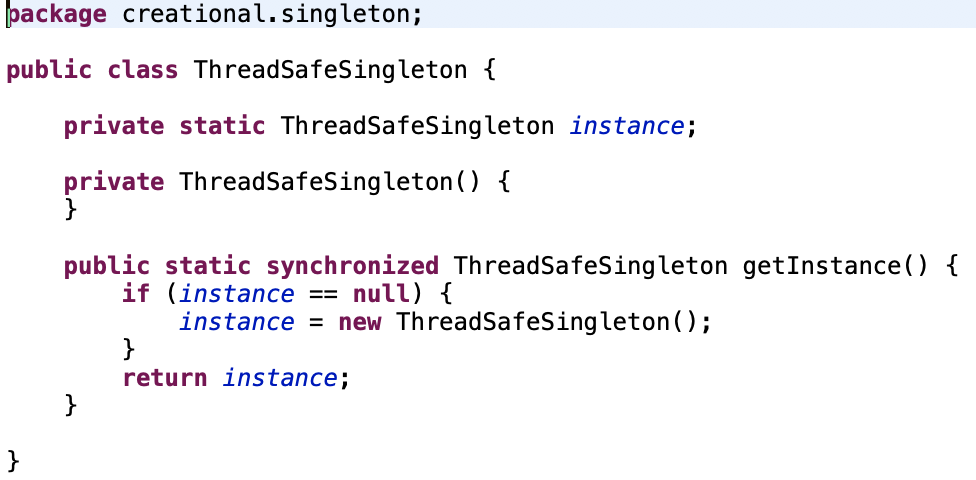
**LazySingleton**

Here the singleton is created when the first call is made.



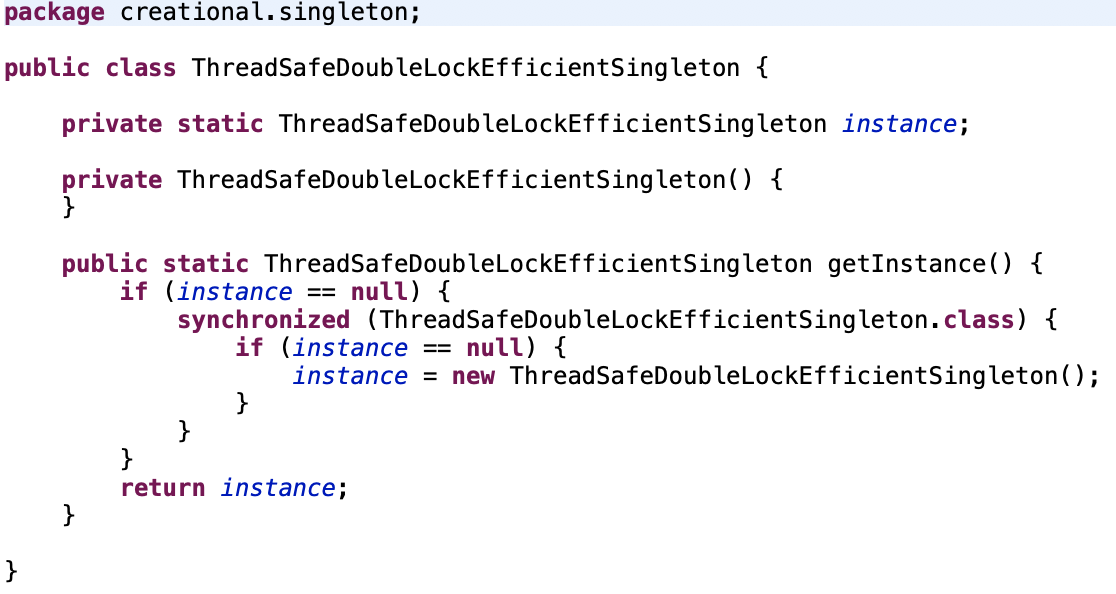
**ThreadSafeSingleton**

The global access point is marked synchronized which causes overhead.

****

**ThreadSafeEfficientSingleton**

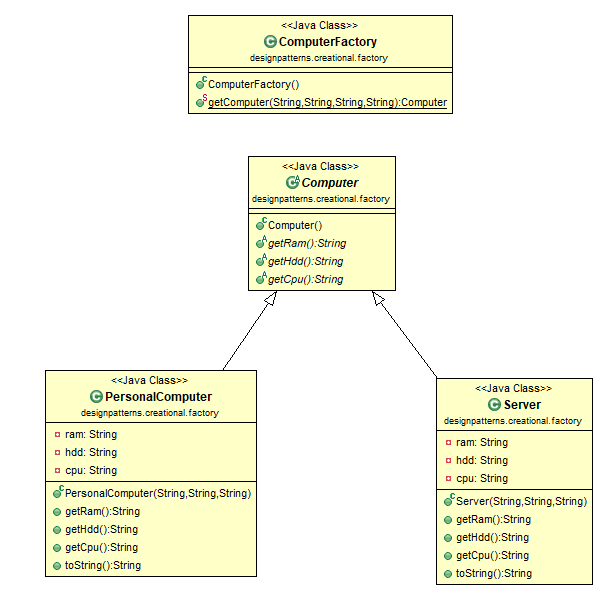
Here first it is checked whether object is already present and then thread locking is done.

****

1. **Factory**

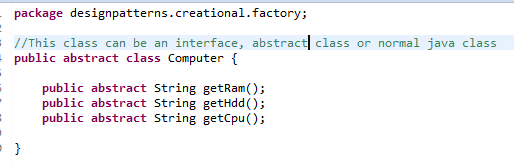
Factory pattern is used when there is one superclass and multiple subclasses. Based on the input given, one subclass is returned. This pattern takes out the responsibility of class instantiation to a factory class. The factory class can be singleton or the factory method can be static.

Example:



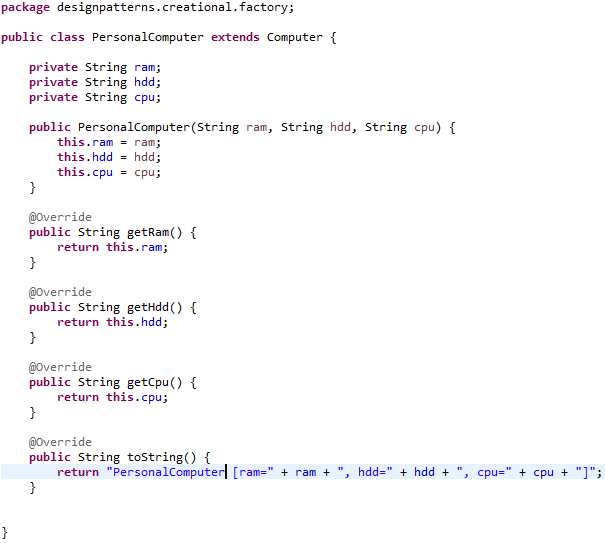
**The Superclass:**

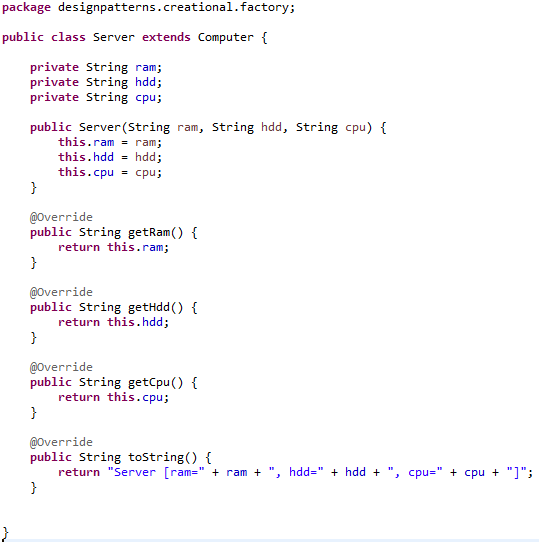
Computer.java – represents any type of computer



**The Subclasses:**

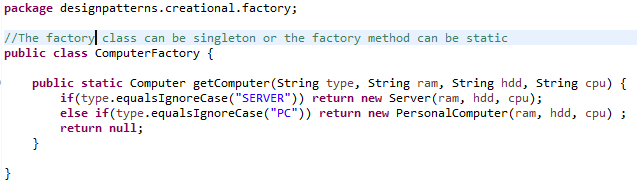
PersonalComputer.java and Server.java which extends Computer.java





**The factory class:**

It uses the superclass computer to get any of the subclasses according to requested type



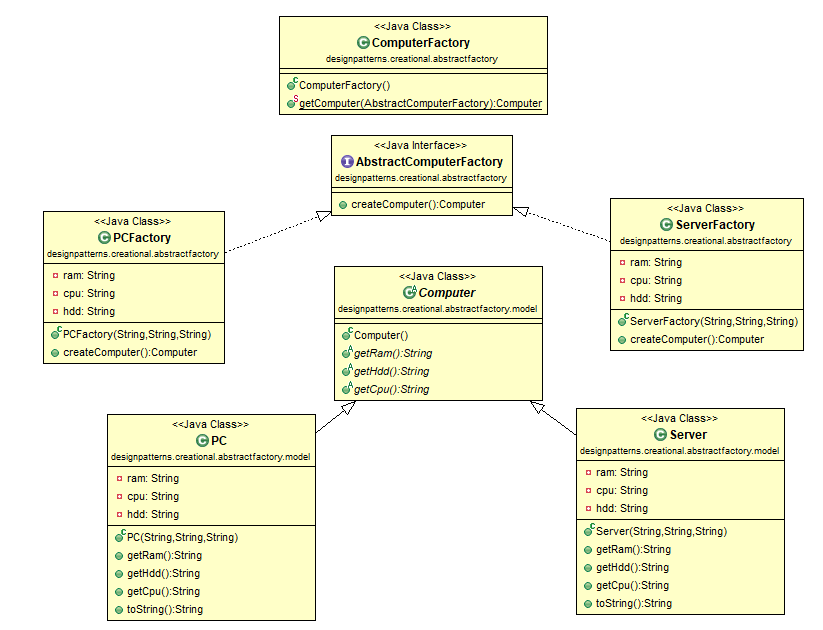
1. **Abstract Factory pattern:**

This is a factory of factories.

The PCFactory creates PCs and the ServerFactory creates servers.

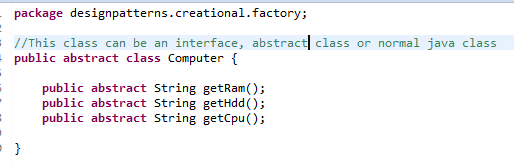
They both implement AbstractComputerFactory.

The ComputerFactory factory method accepts an AbstractComputerFactory which can be either a PCFactory or a ServerFactory at runtime and creates appropriate computers.



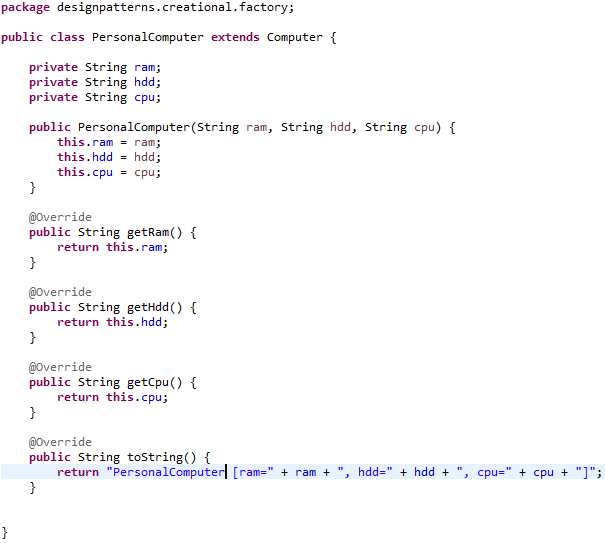
**The Superclass:**

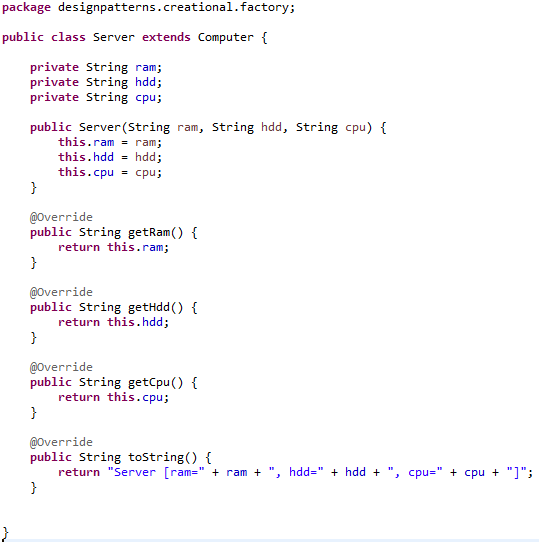
Computer.java – represents any type of computer



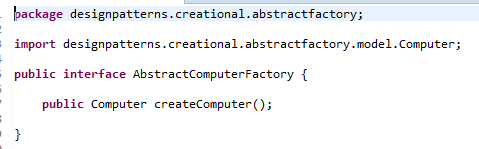
**The Subclasses:**

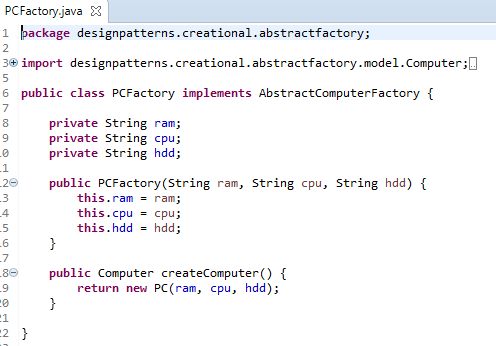
PersonalComputer.java and Server.java which extends Computer.java

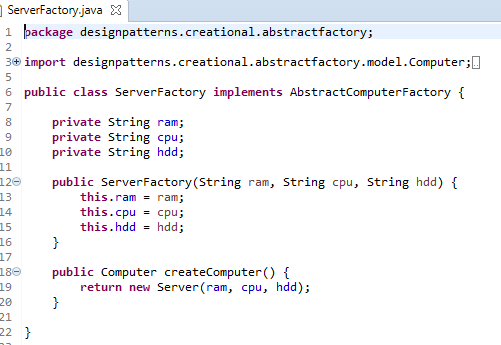




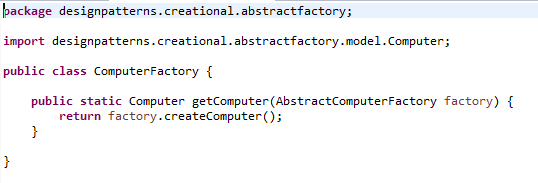
**The abstract factory and its factory implementations:**





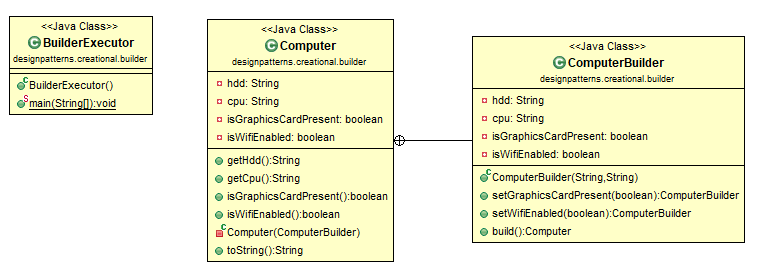


**The factory class which uses the abstract factory:**

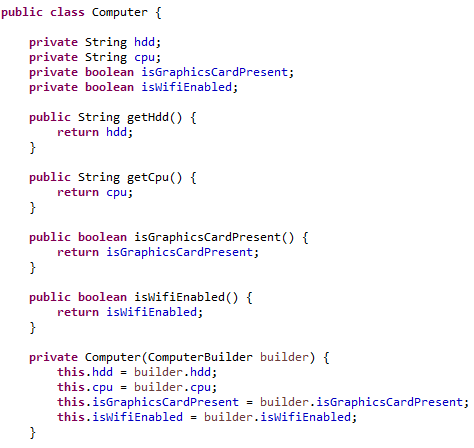


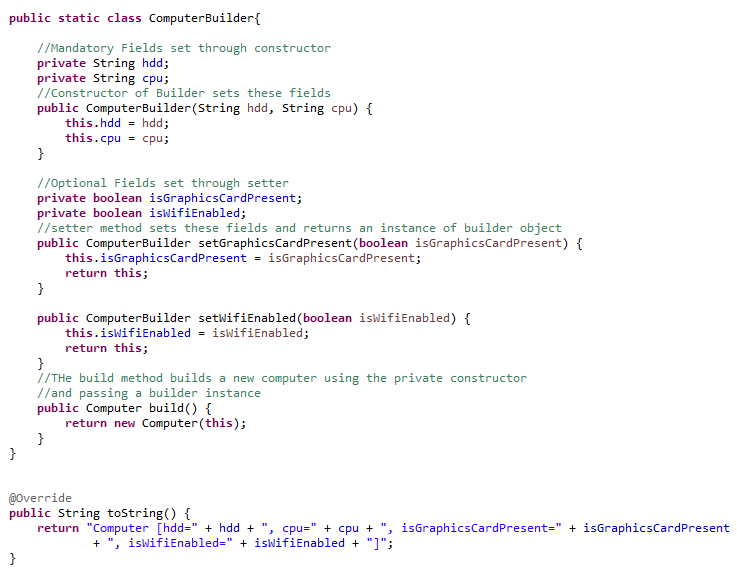
1. **Builder Pattern:**

Builder pattern is used to build complex objects which have mandatory and optional dependencies.

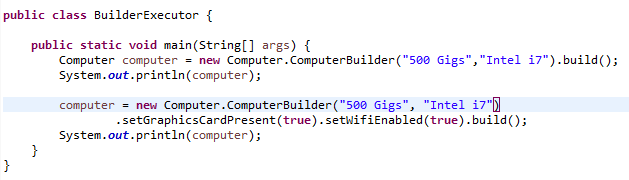


The builderClass is a static class ComputerBuilder which is passed into the private constructor of the class to build (here computer). It is written inside the computer class.





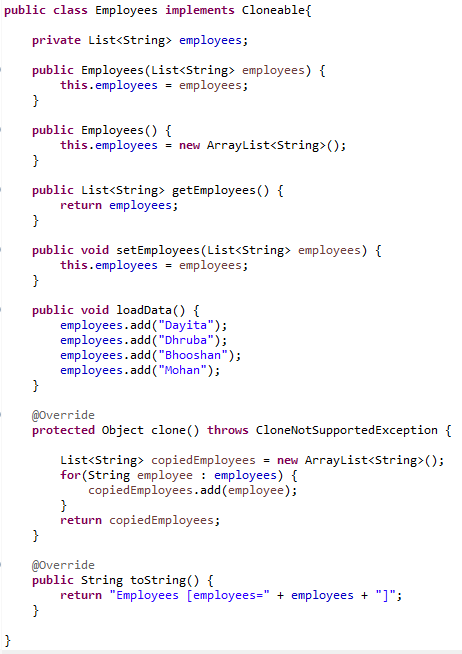
The executor class will do the following.

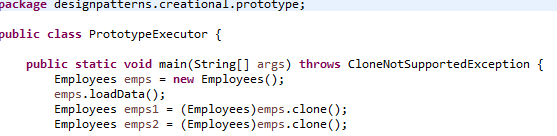


1. **Prototype Pattern**

When object creation is a costly affair and similar objects are available, prototype pattern is used to copy existing objects and then manipulate them.

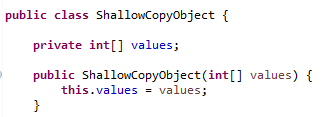
The object to copy implements the cloneable interface and overrides the clone method to achieve this.



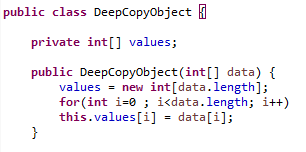


**Deep Copy vs Shallow Copy:**

Shallow copy example:



Deep copy Example:

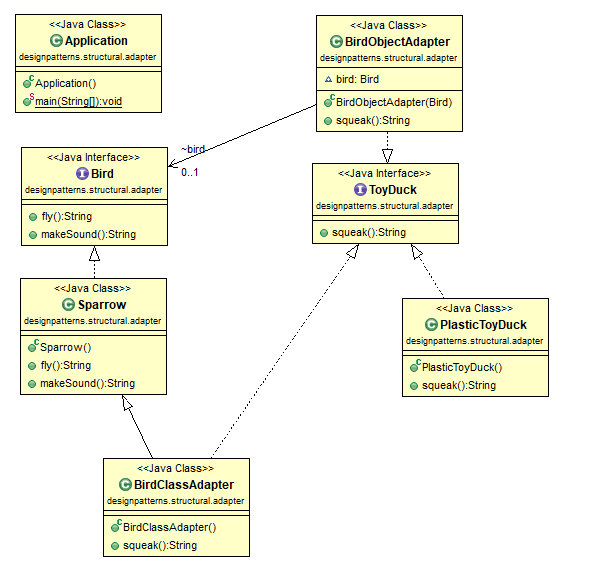


**Structural Design Patterns:**

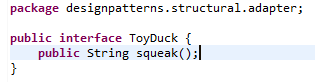
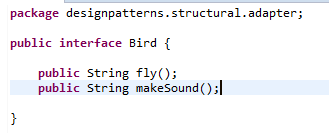
* Adapter
* Composite
* Proxy
* Facade
* Decorator

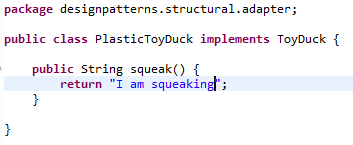
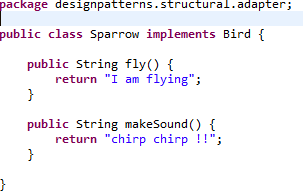
**Adapter Pattern**

Adapter pattern adapts to incompatible interfaces by using an adapter class. It can be implemented as a class or object adapter.



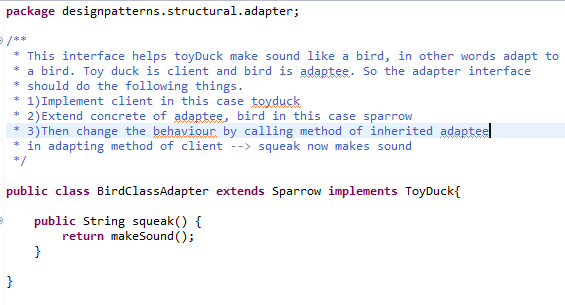
Here we are trying to adapt Toyduck to Bird so that Toyduck can also makeSound.





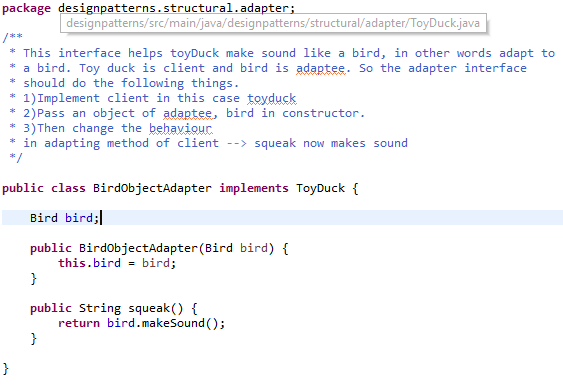
1. BirdClassAdapter adapts ToyDuck to concrete object of Bird interface, Sparrow.

A toyduck can only squeak. But now using the adapter, it can also make sound of a sparrow – Chirp Chirp. This is done using inheritance.

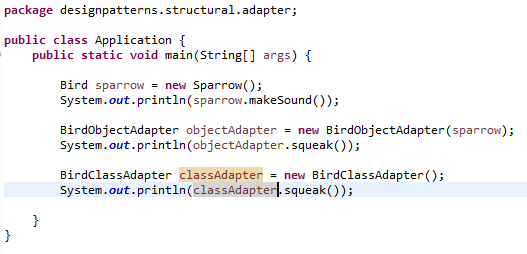


1. BirdObjectAdapter adapts ToyDuck to Bird interface.

A toyduck can only squeak. But now using the adapter, it can also make sound of a sparrow – Chirp Chirp. This is done using composition where there is an actual instance of a bird inside the adapter.

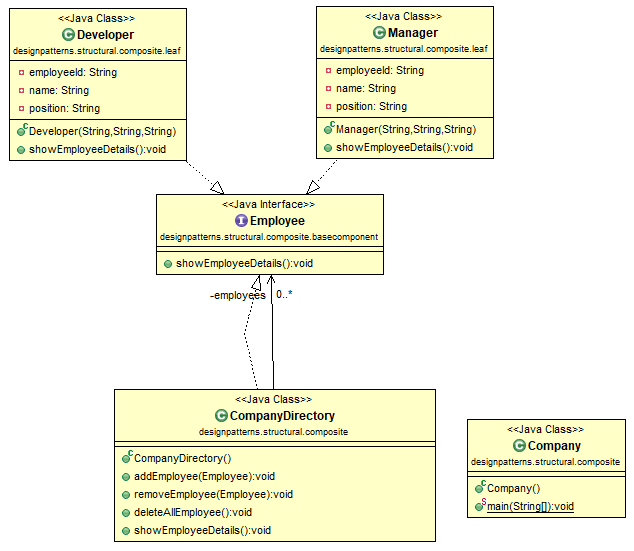


Execution:



**Composite Pattern:**

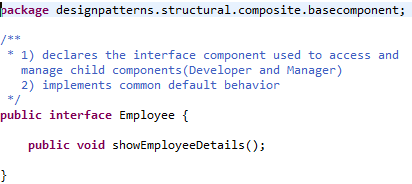
This design pattern treats a group of objects like an instance of a similar object.



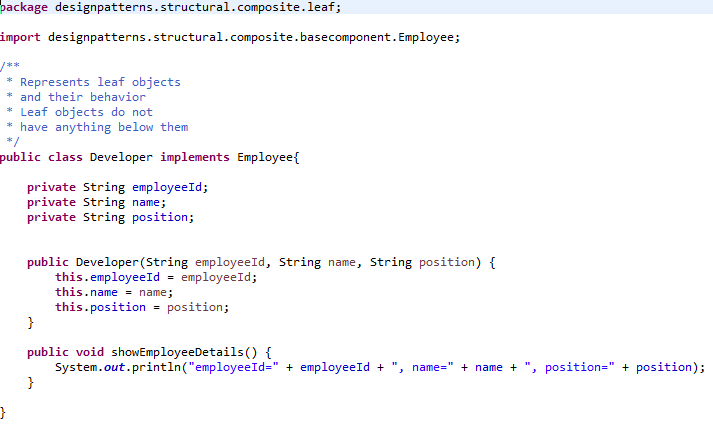
It has 4 parts:

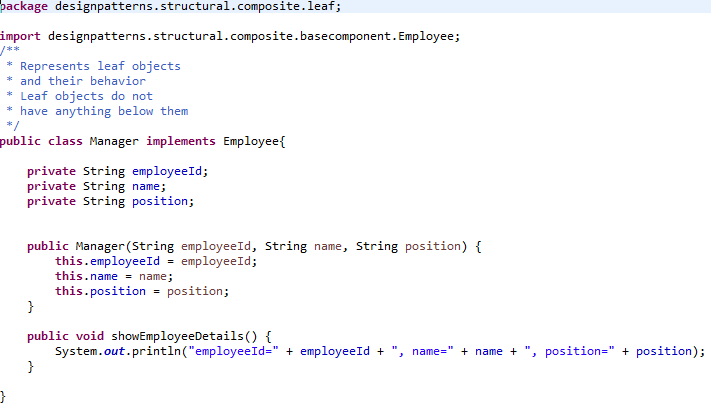
* Base component interface,
* Leaves which implement base component interface,
* Composite which consists of child components and different child operations
* And Client which interacts using the base component

**Base Component Interface:**



**Leaves:**





**Composite:**

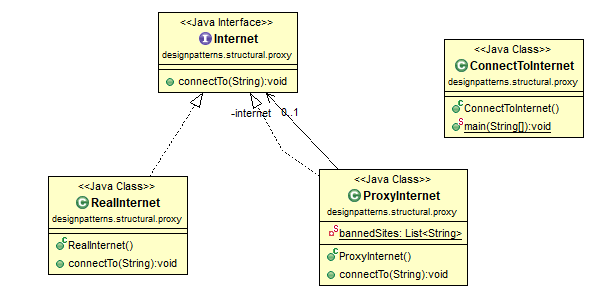


**Client:**



**Proxy Pattern:**

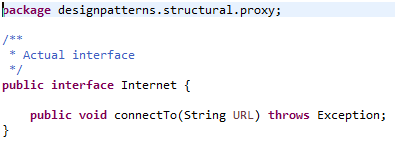
Proxy pattern is used when a class acts as a surrogate or representative for another class.



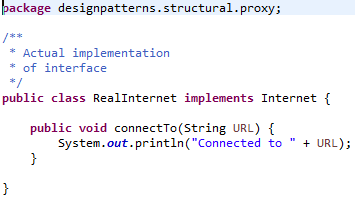
This pattern involves an

* Actual interface
* Class implementing the interface
* Proxy class implementing the interface and calling the real class. This also has the filters.
* Client uses the proxy implementation.

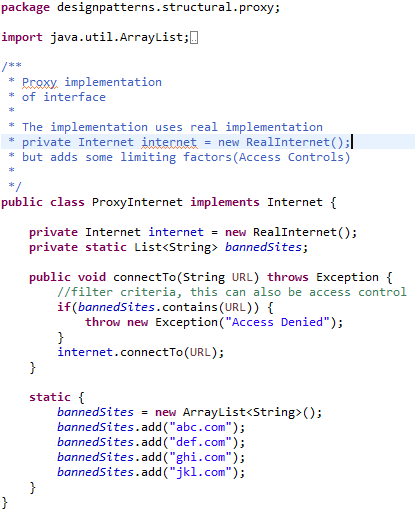
**Actual Interface:**



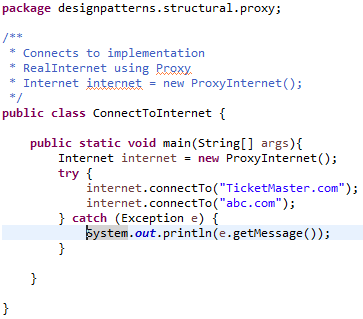
**Actual Class Implementing the interface:**



**Proxy Class implementing the interface, has limit conditions**



**Client**



**Decorator Pattern:**

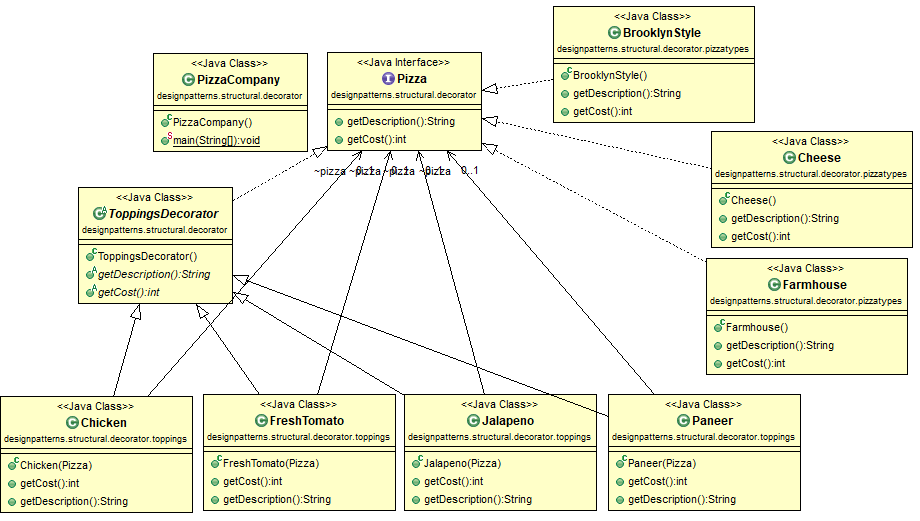
Decorator Pattern adds new features to existing classes without any modification in them.

They follow the open closed principal which states that any class should be open for extension and closed for modification.

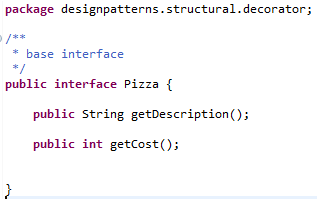
Class instantiation can become complex with multiple decorators in play.

Decorators have the following components:

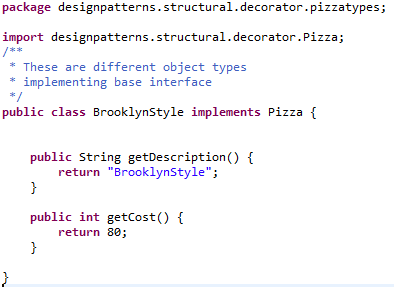
* Base Interface: Pizza
* Implementations of the base interface: Cheese, Farmhouse, and Brooklyn Style
* Decorator class which implements base interface: Toppings Decorator
* Concrete Decorators: Chicken, Fresh Tomato, Jalapeno and Paneer
* Client: Pizza Company

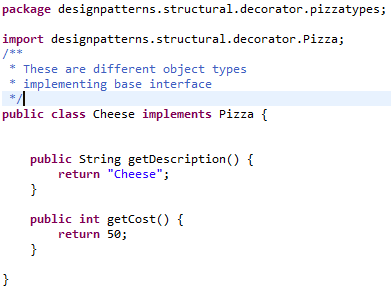


**Base Interface:**

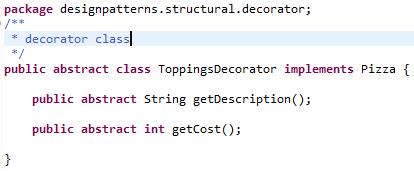


**Implementation of base interface:**

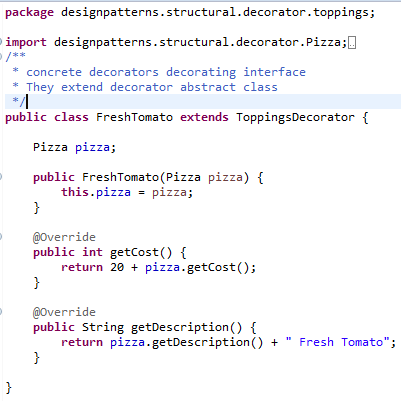
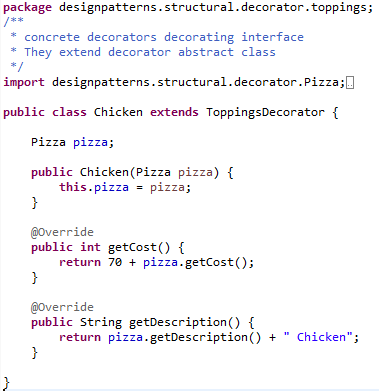




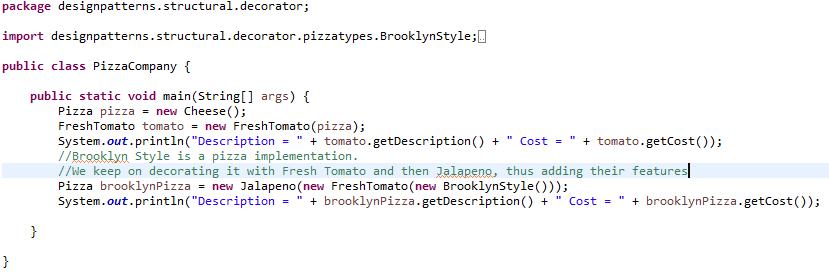
**Decorator:**



**Concrete Decorators:**



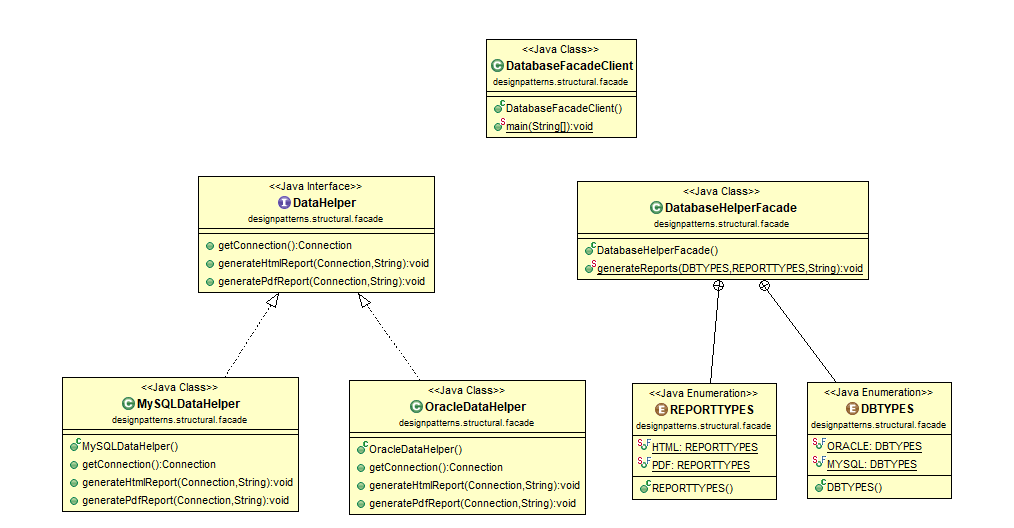
**Client:**



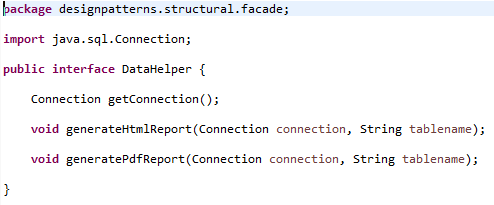
**Facade Pattern:**

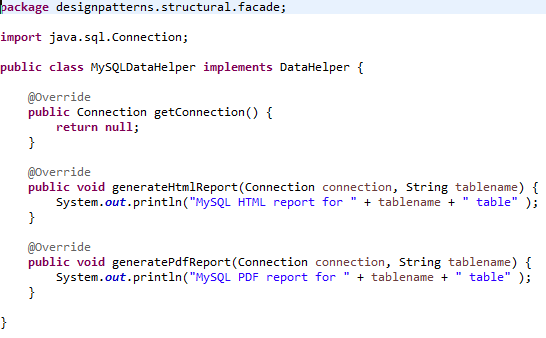
Facade pattern is used to easily interact with a complex set of set of subsystems.

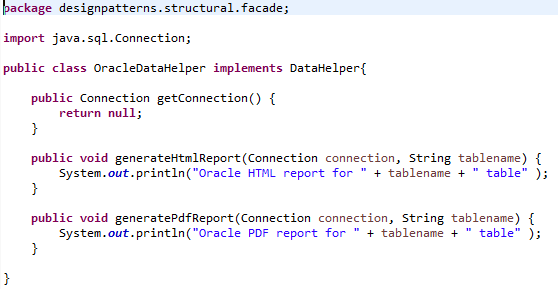
In the following example, Clients will interact through a facade a set of sub systems which comprise of Oracle and MySQL databases.



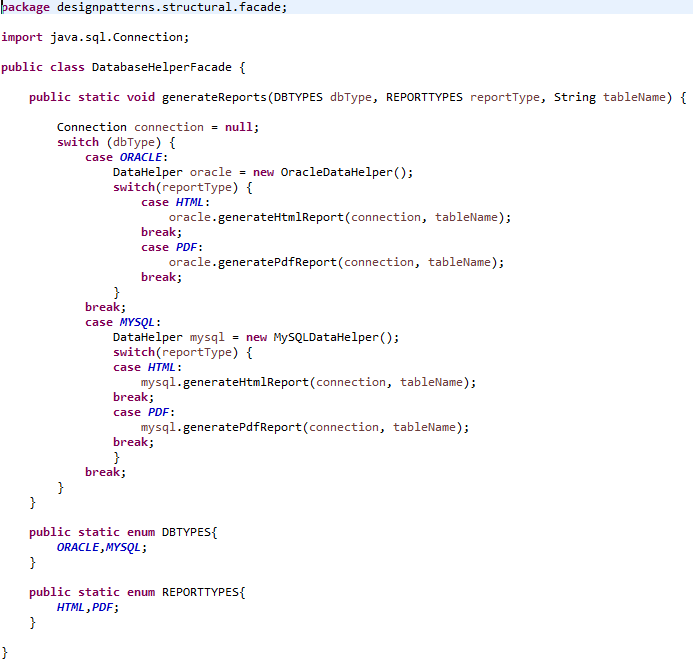
**The set of subsystems ( A set of oracle and MySql databases):**



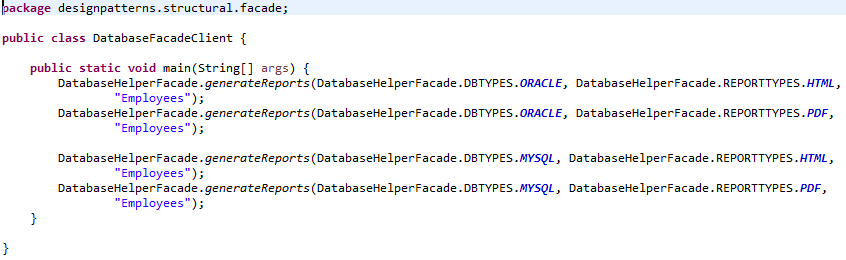




**The facade:**



**The client:**



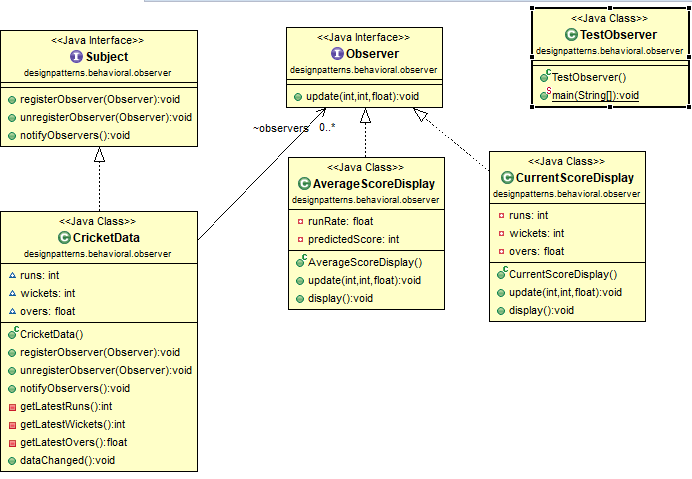
**Behavioral:**

* Observer
* Strategy

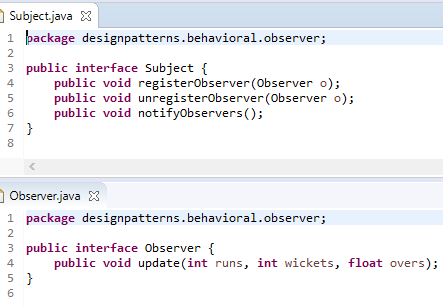
**Observer:**

The observer pattern defines one (subject) to many (observers) dependency between objects so that when one object changes, all its dependents are notified. The observers do not have access to data, they depend on the subject to provide them with data.

* There will be observer interface and subject interface.
* All observers needing data will implement observer interface.
* Notify() in observer will define what needs to be done when subject provides data
* Subject will maintain an Observer Collection which is the collection of registered observers.
* registerObserver(Observer) adds observer and unregisterObserver(observer) removes observer
* notifyObservers() is called when data changes in subject and observers need to be notified

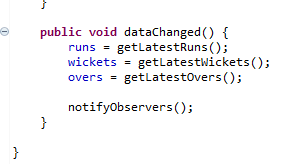


**Observer and Subject Interfaces:**

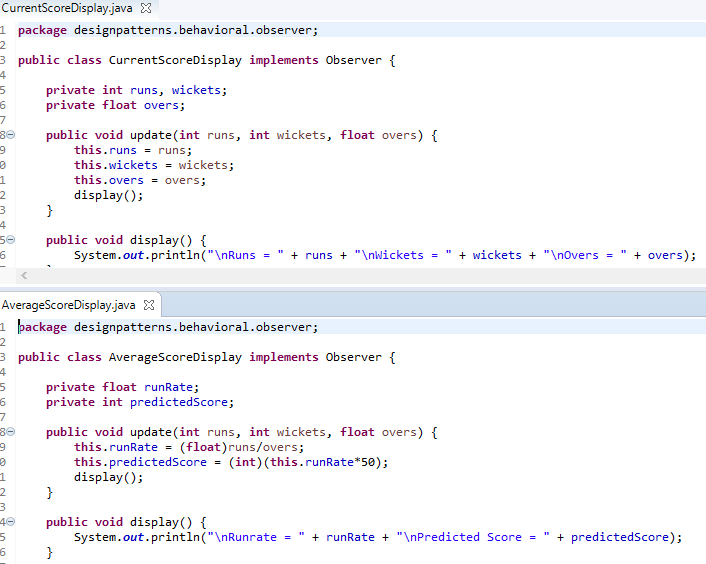


**Subject Implementation:**

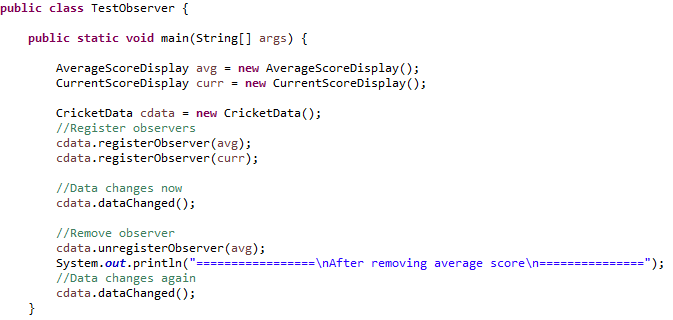




**Observer Implementation:**

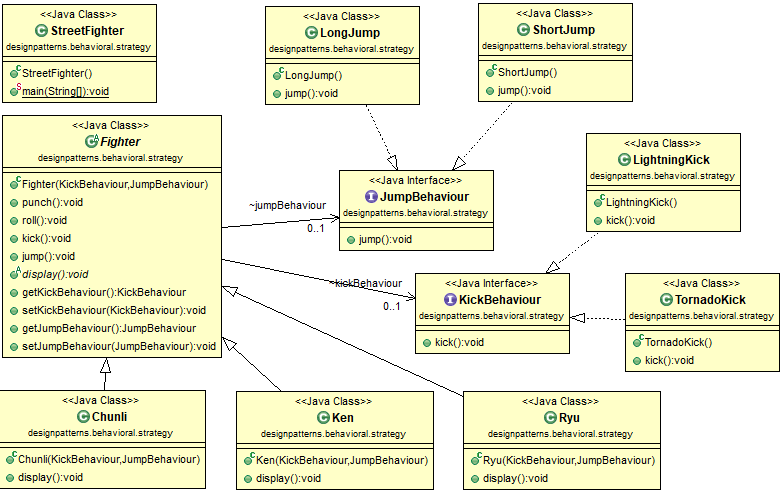


**Executing the observer:**

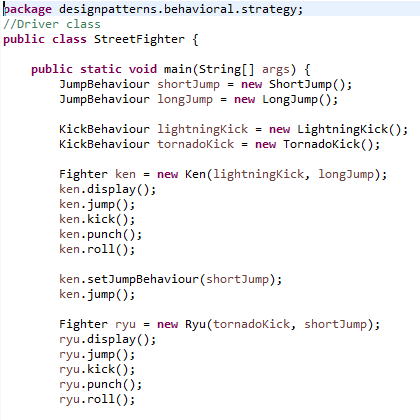


**Strategy Pattern:**

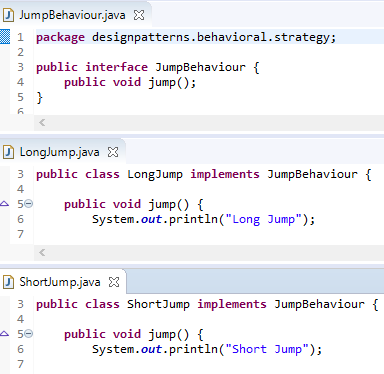
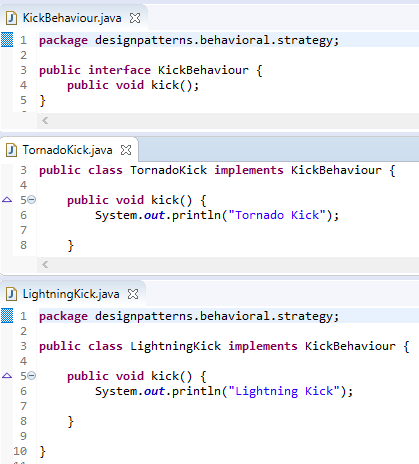
It is a pattern which enables an algorithm’s behavior to be selected at runtime.



**Driver:**

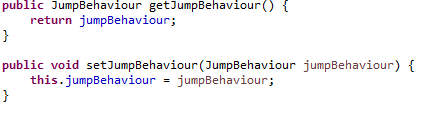


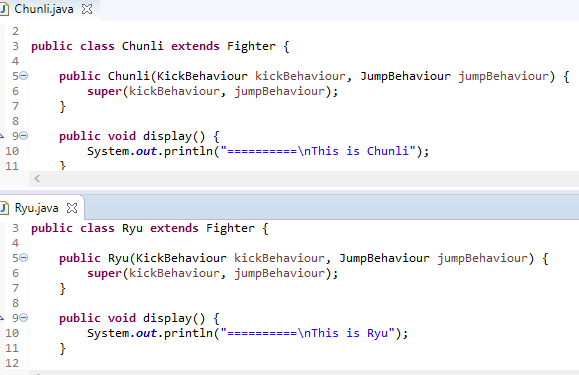
**Behavior:**



**Context:**







**Command Pattern:**