Adapter Pattern
Façade Pattern
Iterator Pattern

514770-1
Fall 2021
11/8/2021
Kyoung Shin Park
Computer Engineering
Dankook University

Adapter Pattern

- "**Convert the interface** of a class into **another interface** clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces."
- Also known as "**Wrapper**"
- Use the adapter pattern when you need to make two classes work with **incompatible interfaces**.

Adapter Pattern

- java.util.Arrays#asList()
- java.util.Collections#list()
- java.util.Collections#enumeration()
- java.io.InputStreamReader(InputStream) (returns a Reader)
- java.io.OutputStreamWriter(OutputStream) (returns a Writer)
- javax.xml.bind.annotation.adapters.XmlAdapter#marshal() and #unmarshal()

Façade Pattern

- "Provide a **unified interface** to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use."
- Provide a **simple interface** to a library, a framework, or one or more complex subsystems.

Facade Pattern

- javax.faces.context.FacesContext, it internally uses among others the abstract/interface types LifeCycle, ViewHandler, NavigationHandler and many more without that the enduser has to worry about it (which are however overrideable by injection).
- javax.faces.context.ExternalContext, which internally uses ServletContext, HttpSession, HttpServletRequest, HttpServletResponse, etc.

Iterator Pattern

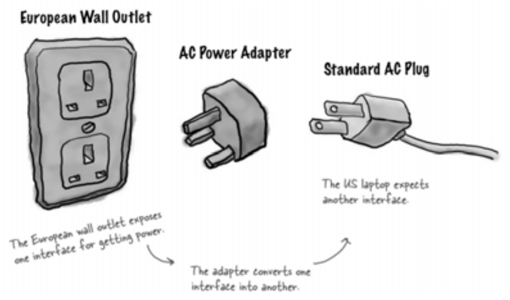
- "Provide a way to **access the elements of an aggregate object sequentially** without exposing its underlying representation."
- Also known as Cursor
- Java iterator design pattern is commonly used in collection framework to traverse through collection objects.
- The C++ and Java standard library abstraction that makes it possible to decouple collection classes and algorithms.

Iterator Pattern

- All implementations of java.util.Iterator
- All implementations of java.util.Enumeration

Adapter Pattern

- Object wrapping
  - Adapter patterns works as a bridge between two incompatible interfaces.
  - This pattern involves a single class which is responsible to join functionalities of independent or incompatible interfaces.
- Example: Electric power plug
  - Different plug (European, US) can be converted using an adapter.


European Wall Outlet
AC Power Adapter
Standard AC Plug
The US laptop expects another interface.
The European wall outlet exposes one interface for getting power.
The adapter converts one interface into another.

Adapter Pattern

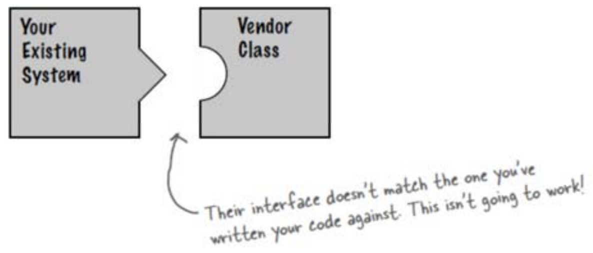
- Object Oriented Adapter
  - OO adapters play the same role as their real world counterparts.
  - They take an interface and adapter it to one that a client is expecting.

Adapter Pattern

	Description
Pattern	Adapter
Problem	Need to make two classes work with incompatible interfaces
Solution	Create an adapter that converts the interface of one object so that another object can understand it.
Result	The adapter helps to handle the changes in the API being called and the client code without changes on either side.

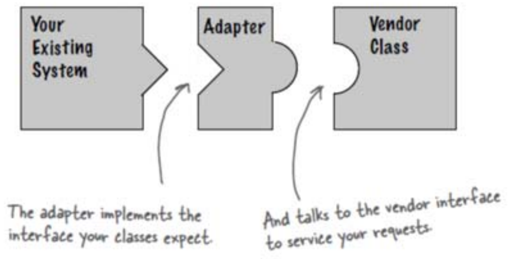
OOP Adapter

- Say you've got an existing software system that you need to work a new vendor class library.
- But, the new vendor designed their interfaces differently than the last vendor.


Your Existing System
Vendor Class
Their interface doesn't match the one you've written your code against. This isn't going to work!

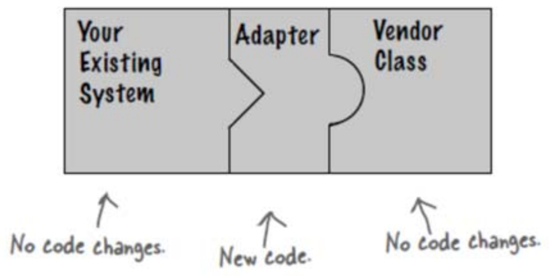
OOP Adapter

- You don't want to solve the problem by changing your existing code (and you can't change the vendor's code). So what do you do?
  - You can write a class that adapts the new vendor interface into the one you're expecting.


Your Existing System
Adapter
Vendor Class
The adapter implements the interface your classes expect.
And talks to the vendor interface to service your requests.

OOP Adapter

- The adapter acts as the middleman by receiving requests from the client and converting them into requests that make sense on the vendor classes.



Duck Magnets (HFDP Ch. 7)

- Duck (HFDP Ch. 1)

```
public interface Duck {
    public void quack();
    public void fly();
}

public class MallardDuck implements Duck {
    public void quack() {
        System.out.println("Quack");
    }
    public void fly() {
        System.out.println("I'm flying");
    }
}
```

Duck Magnets (HFDP Ch. 7)

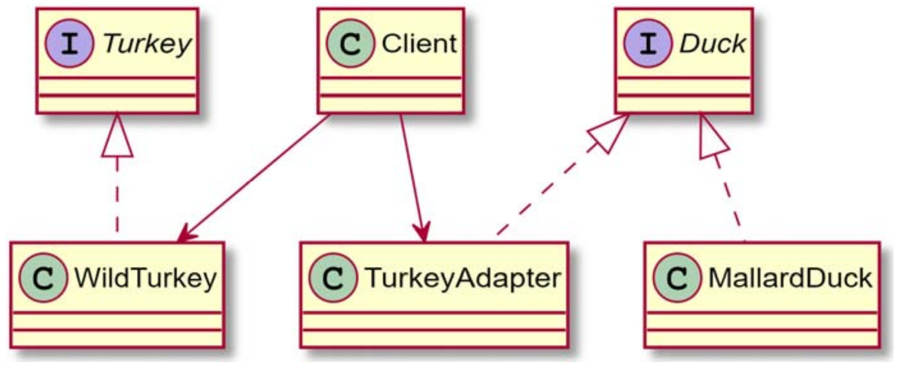
```
public interface Turkey {
    public void gobble();
    public void fly();
}

public class WildTurkey implements Turkey {
    public void gobble() {
        System.out.println("Gobble gobble");
    }
    public void fly() {
        System.out.println("I'm flying a short
distance");
    }
}
```

Duck Magnets (HFDP Ch. 7)

- Let's say you're short on Duck objects and you'd like to use some Turkey objects in their place.
- So, let's write an Adapter

```
public class TurkeyAdapter implements Duck {
    Turkey turkey;
    public TurkeyAdapter(Turkey turkey) {
        this.turkey = turkey;
    }
    public void quack() {
        turkey.gobble();
    }
    public void fly() {
        for (int i = 0; i < 5; i++) {
            turkey.fly();
        }
    }
}
```

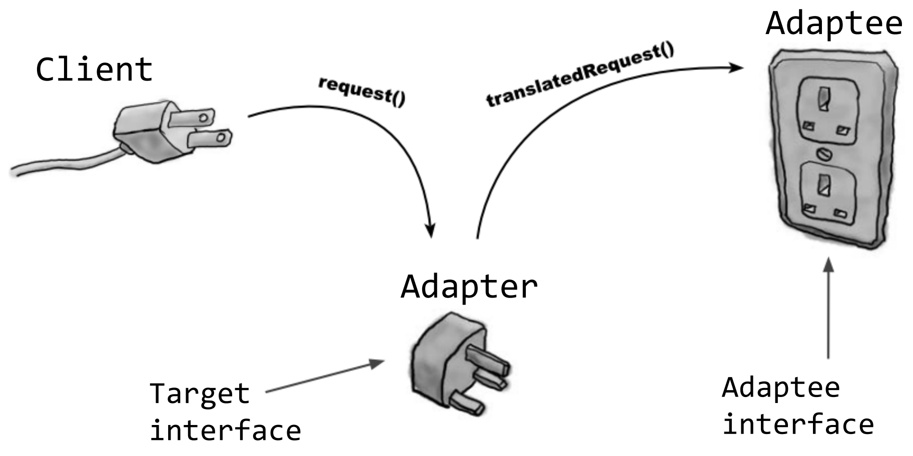
Duck Magnets (HFDP Ch. 7)



Duck Magnets (HFDP Ch. 7)

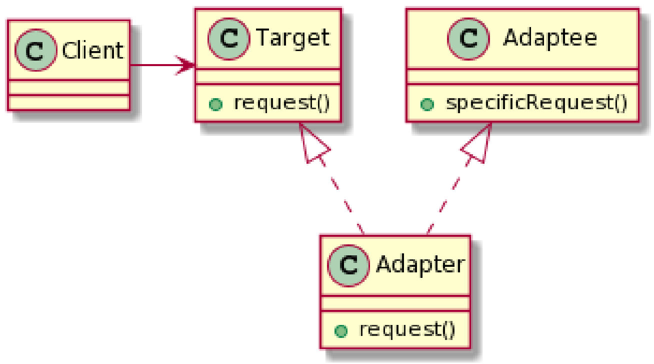
```
public class DuckTestDrive {
  public static void main(String[] args) {
    MallardDuck duck = new MallardDuck();
    WildTurkey turkey = new WildTurkey();
    Duck turkeyAdapter = new TurkeyAdapter(turkey);

    System.out.println("The Turkey says…");
    turkey.gobble();
    turkey.fly();
    System.out.println("\nThe Duck says…");
    testDuck(duck);
    System.out.println("\nThe TurkeyAdapter says…");
    testDuck(turkeyAdapter);
  }
  static void testDuck(Duck duck) {
    duck.quack();
    duck.fly();
  }
}
```

Adapter Pattern



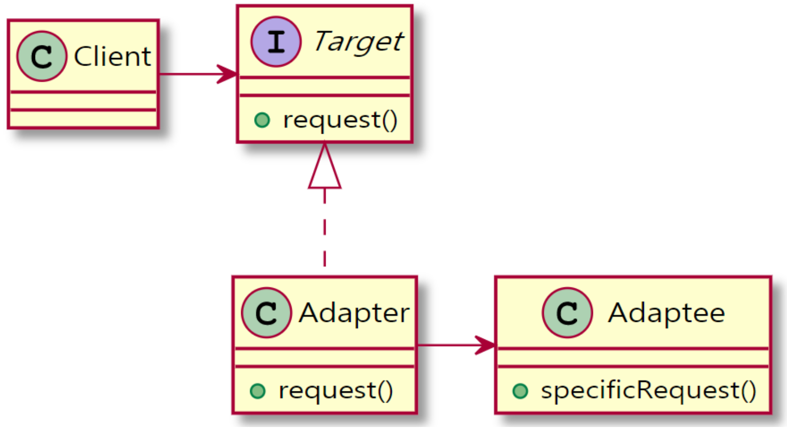
Adapter Pattern

- Class Adapter
  - You need multiple inheritance to implement it, which isn't possible in Java.
  - Class adapter subclasses the Target and the Adaptee.



Adapter Pattern

- Object Adapter
  - Object adapter uses composition to pass requests to an Adaptee.



Define Adapeter Pattern

- Adaptee
  - An object that doesn't support the desired interface.
- Target
  - The interface you want the Adaptee to support
- Adapter
  - The class that makes the Adaptee appear to support the Target interface. Class Adapters use subclass. Object Adapters use composition.

Adapting Enumeration to Iterator (HFDP Ch. 7)

- Enumeration Example

```
import java.util.*;

public class EnumerationExample {
  public static void printEnumeration(Enumeration e) {
    while (e.hasMoreElements()) {
      System.out.println("" + e.nextElement());
    }
  }
  public static void main(String[] args) {
    Vector v = new Vector();
    for (int i = 0; i < 10; i++) {
      v.add(i);
    }
    Enumeration e = v.elements();
    Enumeration1.printEnumeration(e);
  }
}
```

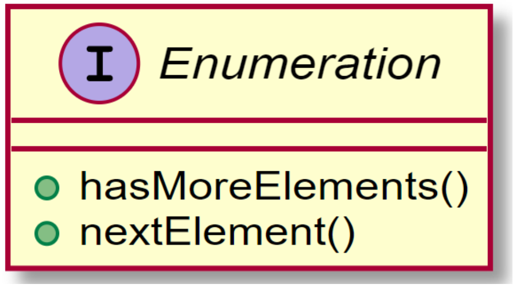
Adapting Enumeration to Iterator (HFDP Ch. 7)

- Iterator Example

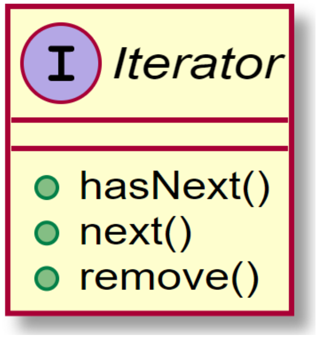
```
import java.util.*;

public class IteratorExample {
  public static void printIterator(Iterator it) {
    while (it.hasNext()) {
      System.out.println("" + it.next());
    }
  }
  public static void main(String[] args) {
    Vector v = new Vector();
    for (int i = 0; i < 10; i++) {
      v.add(i);
    }
    Iterator it = v.iterator();
    Iterator1.printIterator(it);
  }
}
```

Adapting Enumeration to Iterator (HFDP Ch. 7)

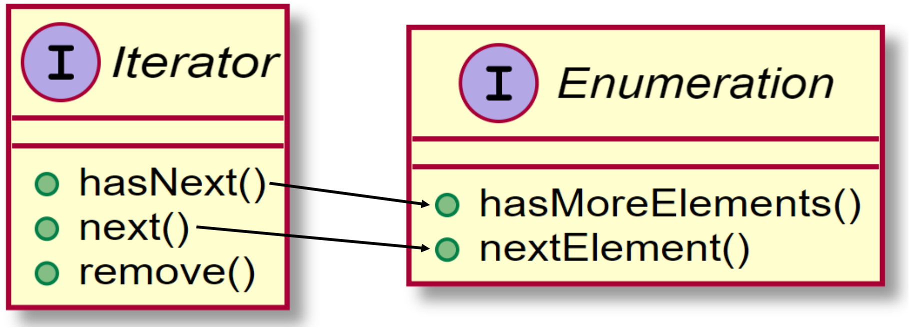
- Enumeration



- Iterator

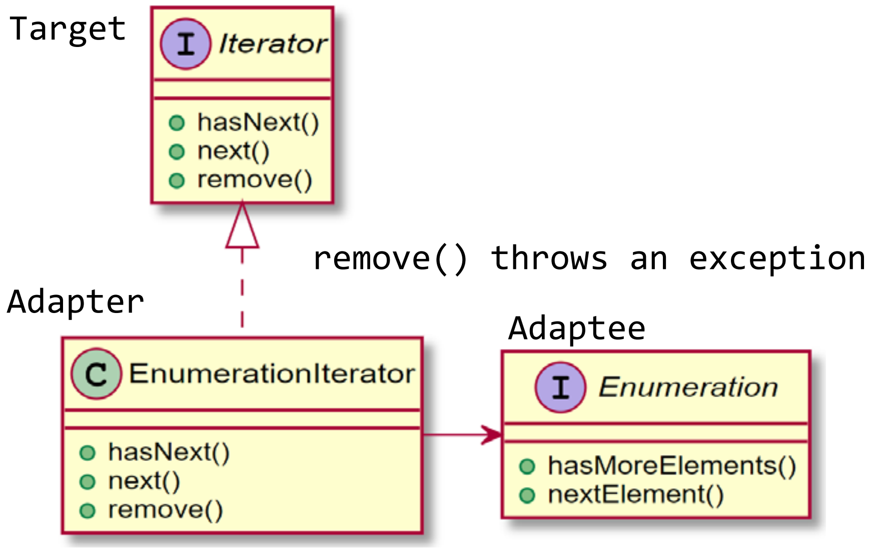


Adapting Enumeration to Iterator (HFDP Ch. 7)



- Iterator – Target interface
- Enumeration – Adaptee interface
- remove() default method

Adapting Enumeration to Iterator (HFDP Ch. 7)



Adapting Enumeration to Iterator (HFDP Ch. 7)

```
public class EnumerationIterator implements Iterator {
  Enumeration enumeration;
  public EnumerationIterator(Enumeration enmt) {
    this.enumeration = enmt;
  }
  public boolean hasNext() {
    return enumeration.hasMoreElements();
  }
  public Object next() {
    return enumeration.nextElement();
  }
  public void remove() {
    throw new UnsupportedOperationException();
  }
}
```

```java
public class Iterator2 {
  public static void printIterator(Iterator it) {
    while (it.hasNext()) {
      System.out.println("" + it.next());
    }
  }

  public static void main(String[] args) {
    Vector v = new Vector();
    for (int i = 0; i < 10; i++) {
      v.add(i);
    }
    Enumeration e = v.elements();
    EnumerationIterator it = new EnumerationIterator(e);
    Iterator2.printIterator(it);
  }
}
```

Arrays Adapter

- **Arrays.asList()** to convert array to immutable list in Java
  - The converted list is an ArrayAdapter and has the characteristics of Array.
  - This converted list cannot use add(), remove() method because it is immutable.
  - This converted list can use set(), get(), contains() method, but ***set() also changes the data in the original array***.

Arrays Adapter

```java
import java.util.Arrays;
import java.util.List;

public class ArraysAdapter {
  public static void main(String[] args) {
    String[] cities = { "Seoul", "Incheon",
"Busan", "Sejong" };
    List<String> cityList =
Arrays.asList(cities);
    System.out.printf("cities.length = %d\n",
cities.length);
    System.out.printf("cityList.size = %d\n",
cityList.size());
```

Arrays Adapter

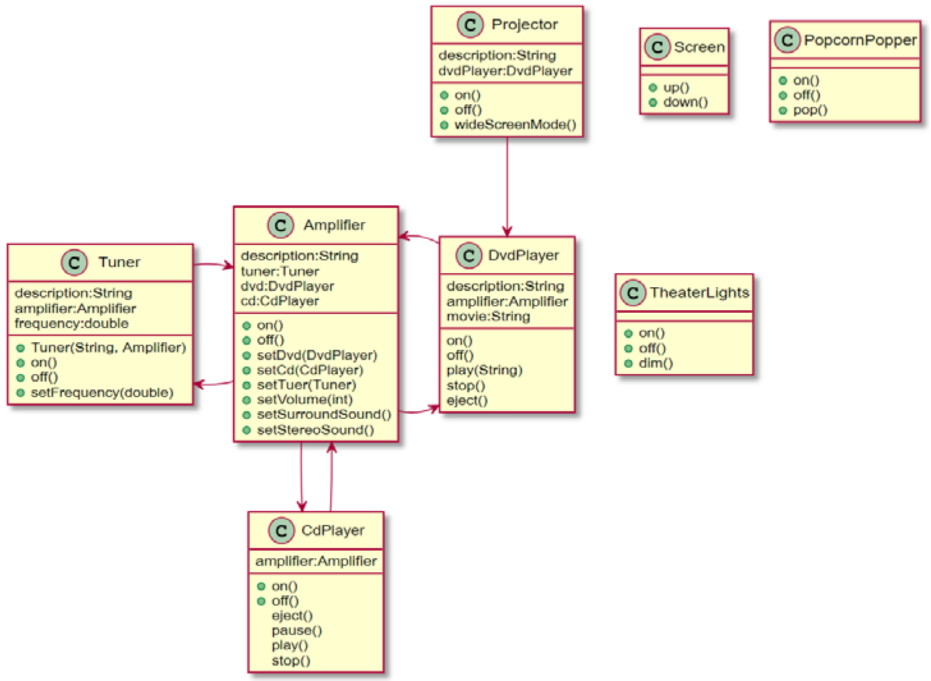
```java
    cityList.set(0, "Suwon");
    System.out.println("\mPrint out cities");
    for (String s : cities) {
      System.out.println(s);
    }
    System.out.println("\mPrint out cityList");
    for (String s : cityList) {
      System.out.println(s);
    }
  }
}
```

Adapter vs Decorator

Pattern	Intent
Decorator	Doesn't alter the interface, but dynamically **adds responsibility** to the interface by wrapping the original code. For example, adding sugar in a coffee.
Adapter	**Converts** one interface to another so that it matches what the client is expecting. For example, electrical adapter.
Facade	Provides a **simplified interface**. Façade not only simplifies an interface, it decouples a client from a subsystem of components.

Home Theater

- Building your own home theater
  - You've assembled a system complete with a DVD player, a projector, an automated screen, surround sound and even a popcorn popper.
- Watching a movie(the hard way)
  - Turn on the popcorn popper, Start the popper popping
  - Dim the lights, Put the screen down
  - Turn the projector on, Set the projector input to DVD
  - Put the projector on wide-screen mode
  - Turn the sound amplifier on, Set the amplifier to DVD input
  - Set the amplifier to surround sound
  - Set the amplifier volume to medium (5)
  - Turn the DVD player on
  - Start the DVD player playing



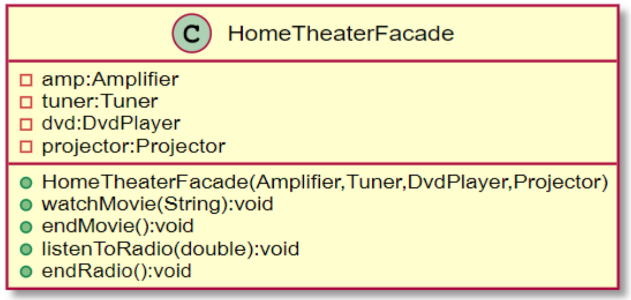
Home Theater

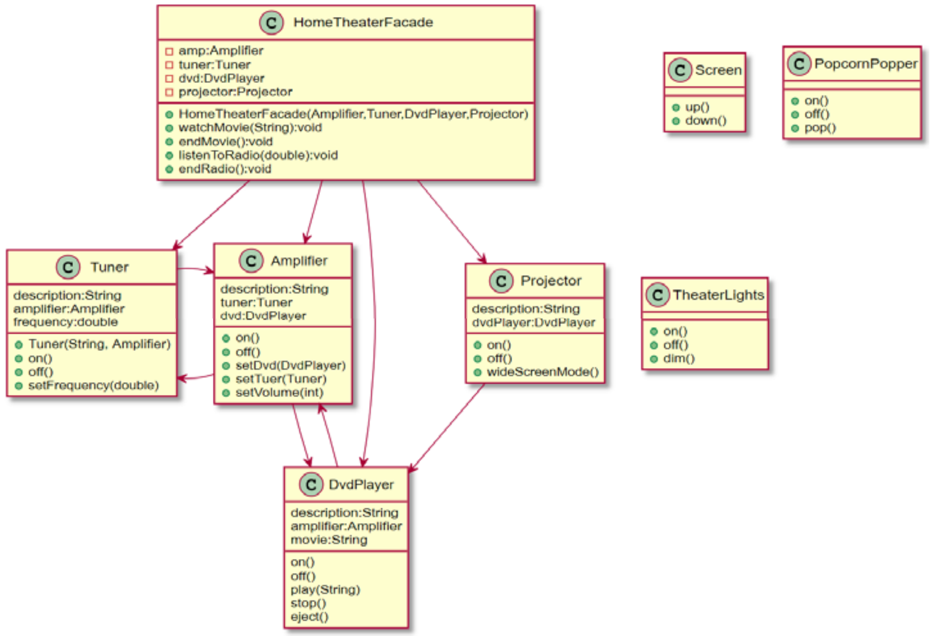
- Watching a movie code

```
popper.on();
popper.pop();
lights.dim(10);
screen.down();
projector.on();
projector.setInput(dvd);
projector.wideScreenMode();
amp.on();
amp.setDvd(dvd);
amp.setSurroundSound();
amp.setVolume(5);
dvd.on();
dvd.play(movie);
```

Home Theater using Façade Pattern

- You can take a complex subsystem and make it easier to use by implementing a **Façade** class that provides one, more reasonable interface.
- If you need the power of the complex subsystem, it's still there for you to use, but if all you need is a straightforward interface, the Façade is there for you.





Home Theater using Façade Pattern

- Create a Façade class for the home theater system, **HomeTheaterFacade**
  - which exposes a few simple methods such as **watchMovie()**
- The Façade class treats the home theater components as a subsystem, and calls on the subsystem to implement its **watchMovie()** method.
- The client code now calls methods on HomeTheaterFacade, not on the subsystem.
- The Façade still leaves the subsystem accessible to used directly.
  - If you need the advanced functionality of the subsystem classes, they are available for your use.

```
public class HomeTheaterFacade {
  private Amplifier amp;
  private Tuner tuner;
  private DvdPlayer dvd;
  private CdPlayer cd;
  private Projector projector;
  private TheaterLights lights;
  private Screen screen;
  private PopcornPopper popper;

  public HomeTheaterFacade(Amplifier a, Tuner t,
      DvdPlayer d, CdPlayer c, Projector p,
      Screen s, TheaterLights l, PopcornPopper pp) {
    this.amp = a; this.tuner = t; this.dvd = d;
    this.cd = c; this.projector = p; this.lights = l;
    this.screen = s; this.popper = pp;
  }
```

Home Theater using Façade Pattern

```
  public void watchMovie(String movie) {
    System.out.println("Get ready to watch a movie...");
    popper.on();
    popper.pop();
    lights.dim(10);
    screen.down();
    projector.on();
    projector.wideScreenMode();
    amp.on();
    amp.setDvd(dvd);
    amp.setVolume(5);
    dvd.on();
    dvd.play(movie);
  }
```

Home Theater using Façade Pattern

```
  public void endMovie() {
    System.out.println("Shutting movie theater down..");
    popper.off();
    lights.on();
    screen.up();
    projector.off();
    amp.off();
    dvd.stop();
    dvd.eject();
    dvd.off();
  }
… // other methods
}
```

Home Theater using Façade Pattern

```
  public void listenToRadio(double frequency) {
    System.out.println("Tuning in the airwaves...");
    tuner.on();
    tuner.setFrequency(frequency);
    amp.on();
    amp.setVolume(5);
    amp.setTuner(tuner);
  }
  public void endRadio() {
    System.out.println("Shutting down the 
tuner...");
    tuner.off();
    amp.off();
  }
}
```

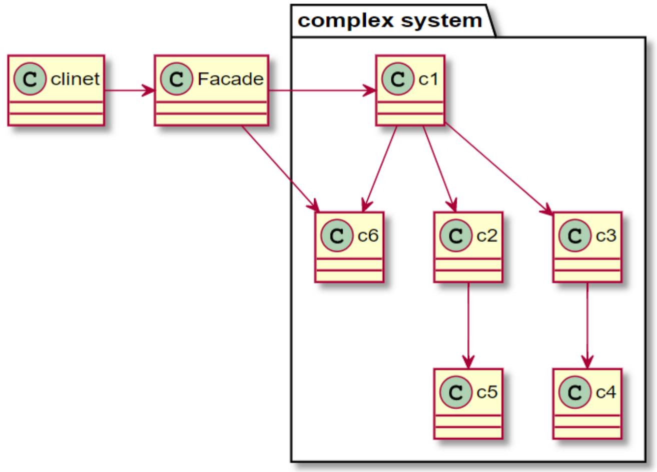
Home Theater using Façade Pattern

```
public class HomeTheaterTestDrive {
  public static void main(String[] args) {
    // create component object
    HomeTheaterFacade homeTheater
     = new HomeTheaterFacade(amp, tuner, dvd, cd,
              projector, screen, lights, popper);
    homeTheater.watchMovie("Raiders of the Lost Ark");
    homeTheater.endMovie();
  }
}
```

Façade Pattern

	Description
Name	Façade
Problem	Clients that access a complex subsystem directly refer to (depend on) many different objects having different interfaces (tight coupling), which makes the clients hard to implement, change, test, and reuse.
Solution	Provide a simple interface to a complex subsystem
Result	Minimize the dependencies on a subsystem. Façade follows **the principle of least knowledge (loose coupling)**.

Façade Pattern



Define Façade Pattern

- Façade
  - Provides a simple interface to a complex subsystem.
- Subsystem Classes
  - Classes that comprise one or more complex subsystems.

Iterator Pattern

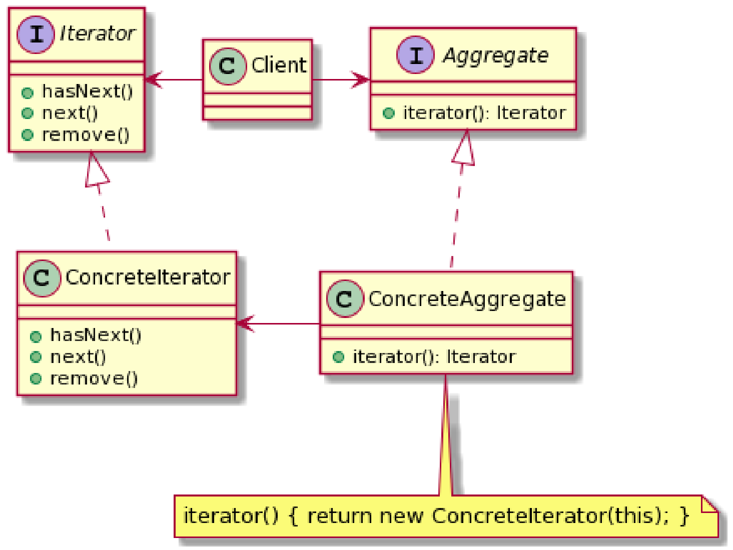
- An aggregate object (such as Array, ArrayList) should give you a way to access its elements without exposing its internal structure.
- Array

```
for (int i = 0; i < arr.length; i++) {
    // do something with arr[i]
    System.out.println(arr[i]);
}
```

- ArrayList

```
for (int i = 0; i < list.size(); i++) {
    // do something with list.get(i)
    System.out.println(list.get(i));
}
```

Iterator Pattern



Define Iterator Pattern

- Iterator
  - An interface to access or traverse the elements collection. Provide methods which concrete iterators must implement.
- ConcreteIterator
  - Implements the Iterator interface methods. It can also keep track of the current position in the traversal of the aggregate collection.
- Aggregate
  - It is typically a collection interface which defines a method that can create an Iterator object.
- ConcreteAggregate
  - It implements the Aggregate interface and its specific method returns an instance of ConcreteIterator.

Iterator Pattern

	Description
Name	Iterator
Problem	Need to “abstract” the traversal of different data structures so that algorithms can be defined that are capable of interfacing with each transparently
Solution	Put the iterator object that defines a standard traversal protocol
Result	Minimize the code change

Iterator

- When you have a set of elements in a collection, you want to **sequentially access those elements**.
- A good example of an Iterator is a TV remote, which has the “next” and “previous” buttons to surf channels.
- If you want to get the average in the list, you need to access each element to calculate sum and average.
- If you want to find a student in the list using ID, you must compare the student’s ID for each element.
- Collection classes have different data structures, which has different methods to access the element.
- When the collection changes, the code that handles each element must also change.

Java Iterator

- Iterator<E> interface is used for iterating (looping) collection classes such as HashMap, ArrayList, LinkedList.
- Iterator<E> method

Method	Description
hasNext()	Check if there are more elements (return true or false)
next()	Get the next element. (return an object)
remove()	Remove the returned element

- Iterable<E> method

Method	Description
iterator()	Return a reference to the iterator (Iterator datatype)

Java Iterator

```
ArrayList<String> cities = new ArrayList<>();
cities.add("Seoul");
cities.add("Tokyo");
cities.add("Washington, D.C.");
Iterator<String> iter = cities.iterator();
while (iter.hasNext()) {
    String s = iter.next();
    System.out.println(s);
}
for (Iterator<String> iter = cities.iterator(); iter.hasNext(); ) {
    String s = iter.next();
    System.out.println(s);
}
for (String city : cities) { // foreach
    System.out.println(city);
}
```

Remove Objects from Collection while Iterating

- ArrayList provides remove(int index) or remove(Object obj). **However, the remove() method is used only when the ArrayList is not iterated.**

```
List<String> list = new ArrayList<>(Arrays.asList("a","b",c","d"));
for (int i = 0; i < list.size(); i++) {
    list.remove(i); // when element is deleted, the list size
              //decreases and the indexes of other elements change
}
for (String s : list) {
     list.remove(s); // ConcurrentModificationException
}
Iterator<String> it = list.iterator();
while (it.hasNext()) {
     String s = it.next(); // next() must be called before remove()
     it.remove();
}
```

Generic

- When you use non-generic ArrayList, it stores the Object datatype.

```
ArrayList list = new ArrayList();
list.add("Seoul");
list.add("Tokyo");
list.add("Washington, D.C.");
List.add(new Integer(10)); // can add Integer to String ArrayList
Iterator it = list.iterator();
while (it.hasNext()) {
     String s = (String)it.next(); // ClassCastException:
java.lang.Integer cannot be cast to java.lang.String
     System.out.println(s);
}
```

Generic

- Creating a Generic class or interface
  - Use generic type to class or interface

```
public class MyClass<T> {
    T val;
    void set(T a) {
        val = a;
    }
    T get() {
        return val;
    }
}
```

Generic class MyClass, with Type parameter T

val's data type is T

  - Declaration generic class reference

```
MyClass<String> s;
List<Integer> li;
Vector<String> vs;
```

Generic

- Type parameter
  - **One uppercase character** between '**<**' and '**>**' is used as a type parameter.
  - Popular type parameters
    - E : means Element, and is often used for elements in a collection
    - T : means Type
    - V : means Value
    - K : means Key
  - Cannot create object of type indicated by type parameter
    - **T a = new T(); // error!!**
    - **T[] a = new T[3]; // error!!**
    - **T[] a = (T[]) new Object[3]; // OK**
  - Type parameters instantiated to actual types later.
  - Any character can be used as a type parameter.

Generic

- Specialization to MyClass<String>

```
public class MyClass<T> {
    T val;
    void set(T a) { val = a; }
    T get() { return val; }
}
```

```
public class MyClass<String> {
    String val;
    void set(String a) {
        val = a;
    }
    String get() {
        return val;
    }
}
```

T is String

Generic

- Specialization
  - Create an object by assigning a specific type to a generic type class, done by the compiler
  - **Error when adding other types** of object
  - Primitive datatype (e.g., int, double, etc) **cannot** be used for type parameters

```
// Specify String to generic type T
MyClass<String> s = new MyClass<String>();
s.set("hello");
System.out.println(s.get()); // "hello"
// Specify Integer to generic type T
MyClass<Integer> n = new MyClass<Integer>();
n.set(5);
System.out.println(n.get()); // 5
n.set("hello");  // compile error
MyClass<int> v = new MyClass<int>(); // compile error (int cannot be used)
```

Inner Class

- Inner class is **a class defined inside of another class**.
  - Can be created as *static* or *non-static*
  - Inner class (and their public fields) are hidden from other classes (encapsulated).
  - Inner objects can access/modify the fields of the outer object (if the inner class is not static).
  - Inner class can be declared *in a method* or *within a entire enclosing class*.

```
public class LinkedList<E> extends AbstractSequentialList<E> {
   private static class ListNode { .. }
   private class ListIterator implements Iterator<E> {
   …
   }
   public Iterator iterator() { return new ListIterator(); }
}
```

Inner Class

- Inner class
  - When **a method of your inner class has the same name as a method in your outer class**, and you want to call the outer class' method from your inner class
    - Java won't let you compile because it can't tell what method you want to call
    - You need to refer to the method of the outer class explicitly
    - **OuterClass.this.outerMethod();**
  - **Public inner classes are visible** outside of the outer class
    - **OuterClass.PublicInnerClass inner = outer.new PublicInnerClass();**
  - You can also make **static inner class**
    - If the inner class **does not access** the outer object
    - Static inner class cannot use instance fields of the outer class
    - **OuterClass.StaticInnerClass si = new OuterClass.StaticInnerClass();**