**Adapter**

**Definition**

Convert the existing interfaces to a new interface to achieve compatibility and reusability of the unrelated classes in one application. Also known as Wrapper pattern.

**Where to use & benefits**

* Try to match an interface(WindowAdapter, etc.)
* Make unrelated classes work together.
* Multiple compatibility.
* Increase transparency of classes.
* Make a pluggable kit.
* Delegate objects.
* Highly class reusable.
* Achieve the goal by inheritance or by composition
* Related patterns include
  + [Proxy](http://www.javacamp.org/designPattern/proxy.html), which provides the same interface as its subject, whereas an adapter provides a different interface to the object it adapts.
  + [Decorator](http://www.javacamp.org/designPattern/decorator.html), which focuses on adding new functions to an object, whereas an adapter coordinates two different objects.
  + [Bridge](http://www.javacamp.org/designPattern/bridge.html), which tries to separate an interface from its implementation and make an object vary independently, whereas an adapter tries to change and cooperate the interface of an object.

**Example**

The famous adapter classes in Java API are WindowAdapter,ComponentAdapter, ContainerAdapter, FocusAdapter, KeyAdapter, MouseAdapter and MouseMotionAdapter.

As you know, WindowListner interface has seven methods. Whenever your class implements such interface, you have to implements all of the seven methods. WindowAdapter class implements WindowListener interface and make seven empty implementation. When you class subclass WindowAdapter class, you may choose the method you want without restrictions. The following give such an example.

public interface Windowlistener {

public void windowClosed(WindowEvent e);

public void windowOpened(WindowEvent e);

public void windowIconified(WindowEvent e);

public void windowDeiconified(WindowEvent e);

public void windowActivated(WindowEvent e);

public void windowDeactivated(WindowEvent e);

public void windowClosing(WindowEvent e);

}

public class WindowAdapter implements WindowListner{

public void windowClosed(WindowEvent e){}

public void windowOpened(WindowEvent e){}

public void windowIconified(WindowEvent e){}

public void windowDeiconified(WindowEvent e){}

public void windowActivated(WindowEvent e){}

public void windowDeactivated(WindowEvent e){}

public void windowClosing(WindowEvent e){}

}

Here is a test program

import javax.swing.\*;

import java.awt.event.\*;

class Test extends JFrame {

public Test () {

setSize(200,200);

setVisible(true);

addWindowListener(new Closer());

}

public static void main(String[] args) {

new Test();

}

class Closer extends WindowAdapter {

public void windowClosing(WindowEvent e) {

System.exit(0);

}

}

}

To reuse classes and make new class compatible with existing ones. For example, A clean system is already designed, you want to add more job in, the Extra interface uses adapter pattern to plug in the existing system.

interface Clean {

public void makeClean();

}

class Office implements Clean{

public void makeClean() {

System.out.println("Clean Office");

}

}

class Workshop implements Clean{

public void makeClean() {

System.out.println("Clean Workshop");

}

}

interface Extra extends Clean{

public void takeCare();

}

class Facility implements Extra{

public void makeClean() {

System.out.println("Clean Facility");

}

public void takeCare() {

System.out.println("Care has been taken");

}

}

In order to reuse Workshop and Office classes,

we create an adapter interface Extra and

add new job takeCare in the system.

class Test {

static void Jobs (Extra job) {

if (job instanceof Clean)

((Clean)job).makeClean();

if (job instanceof Extra)

((Extra)job).takeCare();

}

public static void main(String[] args) {

Extra e = new Facility();

Jobs(e);

Clean c1 = new Office();

Clean c2 = new Workshop();

c1.makeClean();

c2.makeClean();

e.makeClean();

}

}

| C:\ Command Prompt  C:\> java Test  Clean Facility  Care has been taken  Clean Office  Clean Workshop  Clean Facility |
| --- |

By composition, we can achieve adapter pattern. It is also called wrapper. For example, a Data class has already been designed and well tested. You want to adapt such class to your system. You may declare it as a variable and wrapper or embed it into your class.

//well-tested class

class Data {

public void add(Info){}

public void delete(Info) {}

public void modify(Info){}

//...

}

//Use Data class in your own class

class AdaptData {

Data data;

public void add(Info i) {

data.add(i);

//more job

}

public void delete(Info i) {

data.delete(i);

//more job

}

public void modify(Info i) {

data.modify(i);

//more job

}

//more stuff here

//...

}