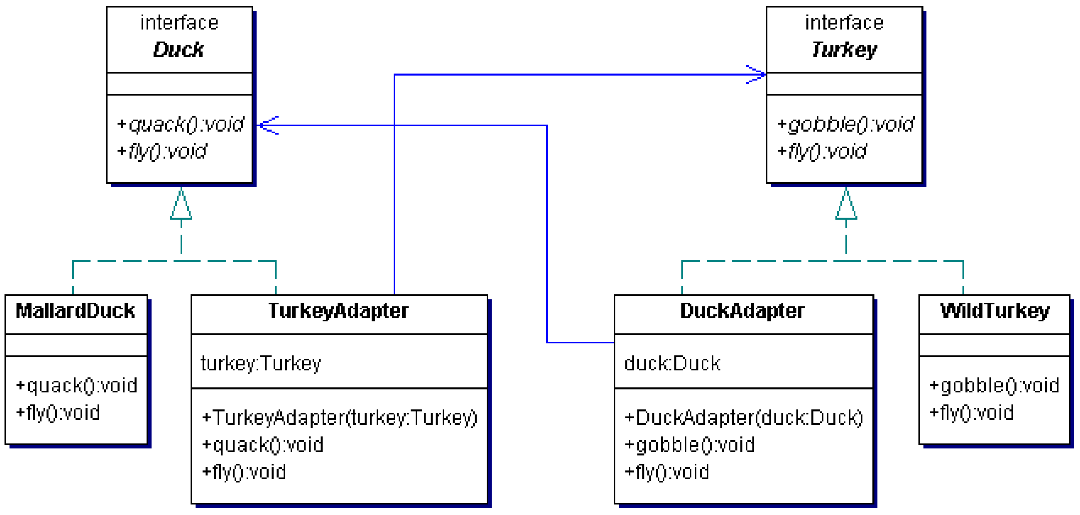
The Adapter Pattern

Lab 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");
   }
}
```

```
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");
   }
}
```

```
public class TurkeyAdapter implements Duck {
   Turkey turkey;

   public TurkeyAdapter(Turkey turkey) {
      this.turkey = turkey;
```

```java
    }

    public void quack() {
        turkey.gobble();
    }

    public void fly() {
        for (int i = 0; i < 5; i++) {
            turkey.fly();
        }
    }
}
```

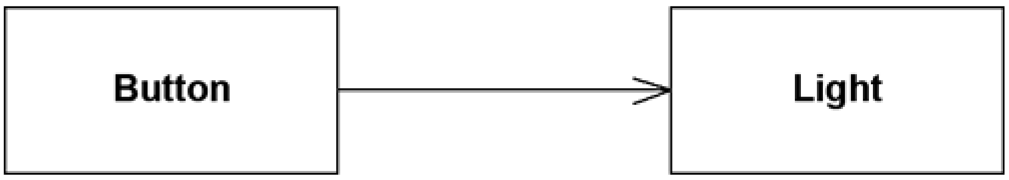
```java
public class DuckAdapter implements Turkey {
    Duck duck;
    Random rand;

    public DuckAdapter(Duck duck) {
        this.duck = duck;
        rand = new Random();
    }

    public void gobble() {
        duck.quack();
    }

    public void fly() {
        if (rand.nextInt(5) == 0) {
            duck.fly();
        }
    }
}
```

```java
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();
    }
}
```

```java
public class TurkeyTestDrive {
   public static void main(String[] args) {
      MallardDuck duck = new MallardDuck();
      Turkey duckAdapter = new DuckAdapter(duck);

      for (int i = 0; i < 10; i++) {
         System.out.println("The DuckAdapter says...");
         duckAdapter.gobble();
         duckAdapter.fly();
      }
   }
}
```

Lab 2:

Consider a `Button` class that uses a `Light` class as follows:

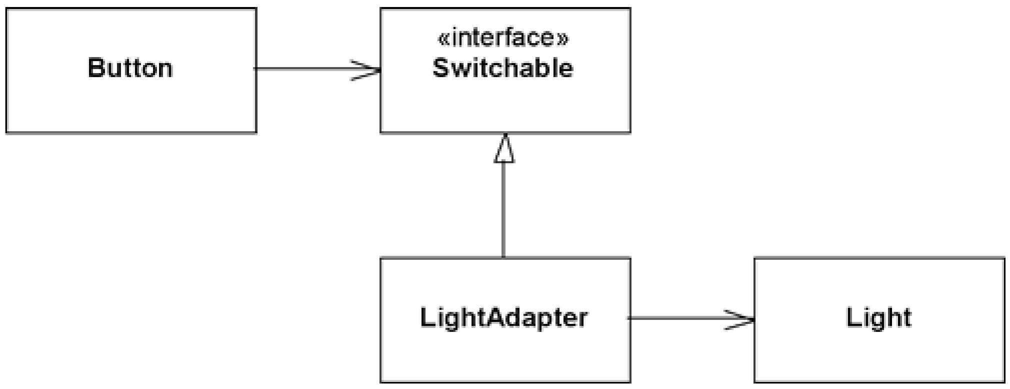


```
public class Button {
   private Light light;

   public Button(Light light) {
      this.light = light;
   }

   public void press() {
      light.turnOn();
   }
}
```

How can be break the dependency between `Button` and `Light`, and thus conform to the SOLID principles of OOD, if we can't modify the `Light` class?

The Adapter Pattern

As shown in the diagram below, this pattern adds an *extra object* called `LightAdapter`, implements the `Switchable` interface and delegates messages received by that interface to the associated `Light` object.



The code that implements the Adapter is trivial.

```java
public class Button {
   Switchable switchable;

   public Button(Switchable switchable) {
      this.switchable = switchable;
   }

   public void press() {
      switchable.turnOn();
   }
}
```

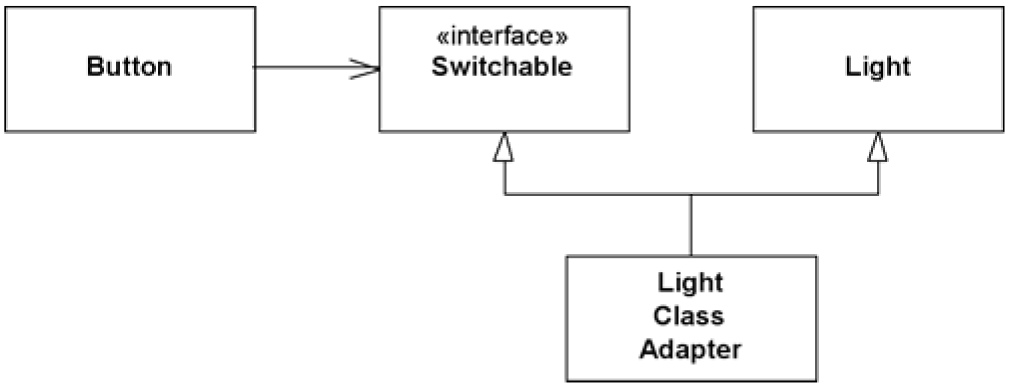
```java
public interface Switchable {
   void turnOn();
}

public class LightAdapter implements Switchable {
   private Light light;

   public LightAdapter(Light light) {
      this.light = light;
   }

   public void turnOn() {
      light.on();
   }
}
```

```java
public class Light {
   private boolean on;
   public void on(){
      System.out.println("Light turns on");
      on = true;
   }

   public void off(){
      System.out.println("Light turns off");
      on = false;
   }

   public boolean isOn() {
      return on;
   }
}
```

This solves the problem nicely. The `Button` class no longer knows about the `Light`, and the `Light` has not been modified.

```
public class LightAdapterTestDrive extends TestCase {
   public void testButtonControlsLight() throws Exception {
      Light l = new Light();
      LightAdapter la = new LightAdapter(l);
      Button b = new Button(la);
      b.press();
      assertTrue(l.isOn());
   }
}
```

The Class Form of Adapter



```
public class LightClassAdapter extends Light implements Switchable {
   public void turnOn() {
      on();
   }
}
```

Isn't it wonderful that this class has no body? And yet, it completely fulfills its role as an Adapter. The unit test below shows how easy this form of the Adapter is to use.

```
public class LightAdapterTestDrive extends TestCase {
  // ...

   public void testButtonControlsLightThroughClassAdapter() {
      LightClassAdapter lca = new LightClassAdapter();
      Button b = new Button(lca);
      b.press();
      assertTrue(lca.isOn());
   }
}
```

Adapter Implemented with Anonymous Inner Class

The anonymous inner class feature of Java can be a wonderful way to create *object*-form Adapters. Consider the following code:

```
public class AdapterAnonymousTest extends TestCase {
   private Light l = new Light();

   public void testAnonymousInnerClassAdapter() throws Exception {
      Switchable s = new Switchable() {
         public void turnOn() {
            l.on();
         }
      };
      Button b = new Button(s);
      b.press();
      assertTrue(l.isOn());
   }
}
```

Adapt a Sender's Protocol

Another use of the Adapter pattern is to *adapt* the protocol of a sender to a receiver. For example, let's say that we have a class that looks like this:

```
public class ThreeWayLight {
   private int brightness = 0;

   public void lo() {
      brightness = 1;
   }

   public void medium() {
      brightness = 2;
   }

   public void high() {
      brightness = 3;
   }

   public void off() {
      brightness = 0;
   }

   public int getBrightness() {
      return brightness;
   }
}
```

The `Button` was never designed to control a three-way light. Moreover, it looks as though the `ThreeWayLight` class was not designed to take input from a `Button`. How can we adapt the `Button` class to the `ThreeWayLight` ? Consider the following unit test:

```
   public void testThreeWayLight() throws Exception {
        ThreeWayLight twl= new ThreeWayLight();
        ThreeWayAdapter twa = new ThreeWayAdapter(twl);
        Button b = new Button(twa);
        assertEquals(0, twl.getBrightness());
        b.press();
        assertEquals(1, twl.getBrightness());
        b.press();
        assertEquals(2, twl.getBrightness());
        b.press();
        assertEquals(3, twl.getBrightness());
        b.press();
        assertEquals(0, twl.getBrightness());
      }
```

Clearly our intent is that the `ThreeWayAdapter` should ratchet the `ThreeWayLight` through its states every time the `Button` is pressed. We can easily implement this adapter as follows:

```
public class ThreeWayAdapter implements Switchable {
   private ThreeWayLight twl;

   public ThreeWayAdapter(ThreeWayLight twl) {
      this.twl = twl;
   }

   public void turnOn() {
      switch (twl.getBrightness()) {
      case 0:
         twl.lo();
         break;
      case 1:
         twl.medium();
         break;
      case 2:
         twl.high();
         break;
      case 3:
         twl.off();
         break;
      }
   }
}
```

Lab 3

```
interface Contact {
   public String getFirstName();

   public String getLastName();

   public String getTitle();

   public String getOrganization();

   public void setContact(Chovnatlh newContact);

   public void setFirstName(String newFirstName);
```

```java
   public void setLastName(String newLastName);

   public void setTitle(String newTitle);

   public void setOrganization(String newOrganization);

}
```

```java
class ContactAdapter implements Contact {
   private Chovnatlh contact;

   public ContactAdapter() {
      contact = new ChovnatlhImpl();
   }

   public ContactAdapter(Chovnatlh newContact) {
      contact = newContact;
   }

   public String getFirstName() {
      return contact.tlhapWa$DIchPong();
   }

   public String getLastName() {
      return contact.tlhapQavPong();
   }

   public String getTitle() {
      return contact.tlhapPatlh();
   }

   public String getOrganization() {
      return contact.tlhapGhom();
   }

   public void setContact(Chovnatlh newContact) {
      contact = newContact;
   }

   public void setFirstName(String newFirstName) {
      contact.cherWa$DIchPong(newFirstName);
   }

   public void setLastName(String newLastName) {
      contact.cherQavPong(newLastName);
   }

   public void setTitle(String newTitle) {
      contact.cherPatlh(newTitle);
   }

   public void setOrganization(String newOrganization) {
      contact.cherGhom(newOrganization);
   }

   public String toString() {
      return contact.toString();
   }
}
```

```java
interface Chovnatlh {
   public String tlhapWa$DIchPong();

   public String tlhapQavPong();

   public String tlhapPatlh();

   public String tlhapGhom();

   public void cherWa$DIchPong(String chu$wa$DIchPong);

   public void cherQavPong(String chu$QavPong);

   public void cherPatlh(String chu$patlh);

   public void cherGhom(String chu$ghom);
}
```

```java
// pong = name
// wa'DIch = first
// Qav = last
// patlh = rank (title)
// ghom = group (organization)
// tlhap = take (get)
// cher = set up (set)
// chu' = new
// chovnatlh = specimen (contact)
```

```java
class ChovnatlhImpl implements Chovnatlh {
   private String wa$DIchPong;

   private String QavPong;

   private String patlh;

   private String ghom;

   public ChovnatlhImpl() {
   }

   public ChovnatlhImpl(String chu$wa$DIchPong, String chu$QavPong,
         String chu$patlh, String chu$ghom) {
      wa$DIchPong = chu$wa$DIchPong;
      QavPong = chu$QavPong;
      patlh = chu$patlh;
      ghom = chu$ghom;
   }

   public String tlhapWa$DIchPong() {
      return wa$DIchPong;
   }

   public String tlhapQavPong() {
      return QavPong;
   }

   public String tlhapPatlh() {
      return patlh;
   }

   public String tlhapGhom() {
      return ghom;
   }

   public void cherWa$DIchPong(String chu$wa$DIchPong) {
      wa$DIchPong = chu$wa$DIchPong;
   }

   public void cherQavPong(String chu$QavPong) {
      QavPong = chu$QavPong;
   }

   public void cherPatlh(String chu$patlh) {
      patlh = chu$patlh;
   }

   public void cherGhom(String chu$ghom) {
      ghom = chu$ghom;
   }

   public String toString() {
      return wa$DIchPong + " " + QavPong + ": " + patlh + ", " + ghom;
   }
}
```

```
public class RunAdapterPattern {
   public static void main(String[] arguments) {
      System.out.println("Example for the Adapter pattern");
      System.out.println();
      System.out
            .println("This example will demonstrate the Adapter by using the");
      System.out
            .println(" class ContactAdapter to translate from classes written");
      System.out
            .println(" in a foreign language (Chovnatlh and ChovnatlhImpl),");
      System.out
            .println(" enabling their code to satisfy the Contact interface.");
      System.out.println();

      System.out
            .println("Creating a new ContactAdapter. This will, by extension,");
      System.out
            .println(" create an instance of ChovnatlhImpl which will provide");
      System.out.println(" the underlying Contact implementation.");
      Contact contact = new ContactAdapter();
      System.out.println();

      System.out.println("ContactAdapter created. Setting contact data.");
      contact.setFirstName("Thomas");
      contact.setLastName("Williamson");
      contact.setTitle("Science Officer");
      contact.setOrganization("W3C");
      System.out.println();

      System.out
            .println("ContactAdapter data has been set. Printing out Contact
data.");
```

```
        System.out.println();
        System.out.println(contact.toString());
    }
}
```