Example of using an interface.

```
/** Encryption.java -- Interface that defines what operations we want
 *  in order to support encryption.  Namely, we need to have a way
 *  to encrypt and decrypt data (Strings).
 */
public interface Encryption
{
  public void encrypt();
  public void decrypt();
}
```

```
/** Encrypt and decrypt data using the Caesar cipher.
 *  For the purpose of this simple program, it's not necessary to keep
 *  around both the plaintext and ciphertext Strings, but if we enhance
 *  the program, they may be useful.
 *
 *  The 'key' attribute is the number we need to add to each letter to
 *  encrypt it.
 */

public class Caesar implements Encryption
{
  private String plaintext;
  private String ciphertext;
  private static int key = 7;

  // no need to initialize data yet
  public Caesar()
  {
  }

  public void setPlaintext(String s)
  {
    plaintext = s;
  }
  public void setCiphertext(String s)
  {
    ciphertext = s;
  }

  /** encrypt -- Go through each character of the string and add the key
   *  value.  Note that the cast is needed because the + operator returns
   *  an integer result.
   */
  public void encrypt()
  {
    ciphertext = "";
    for(int i = 0; i < plaintext.length(); ++i)
      ciphertext += (char) (plaintext.charAt(i) + key);
    System.out.println(ciphertext);
  }

  /** decrypt -- this is exactly the inverse of encrypt
   */
  public void decrypt()
  {
    plaintext = "";
    for (int i = 0; i < ciphertext.length(); ++i)
      plaintext += (char) (ciphertext.charAt(i) - key);
    System.out.println(plaintext);
  }
}
```

```
/** Encrypt and decrypt data using the Xor cipher.
 *  Xor stands for "exclusive or".  This is a technique that manipulates
 *  the binary representation of the characters.  One interesting feature
 *  is that the encryption and decryption functions are the same.
 *  And, like the Caesar cipher, there is a 'key' value we perform the
 *  operation with.
 */

public class Xor implements Encryption
{
  private String plaintext;
  private String ciphertext;
  private static int key = 7;

  // no need to initialize data yet
  public Xor()
  {
  }

  public void setPlaintext(String s)
  {
    plaintext = s;
  }
  public void setCiphertext(String s)
  {
    ciphertext = s;
  }

  /** encrypt -- Go through each character of the string and add the key
   *  value.
   */
  public void encrypt()
  {
    ciphertext = "";
    for(int i = 0; i < plaintext.length(); ++i)
      ciphertext += (char) (plaintext.charAt(i) ^ key);
    System.out.println(ciphertext);
  }

  /** decrypt -- this is exactly the inverse of encrypt
   */
  public void decrypt()
  {
    plaintext = "";
    for (int i = 0; i < ciphertext.length(); ++i)
      plaintext += (char) (ciphertext.charAt(i) ^ key);
    System.out.println(plaintext);
  }
}
```

```
/** Driver.java -- main module for our encryption program
 */

import java.io.*;           // where BufferedReader is defined

public class Driver
{
  public static void main(String [] args) throws IOException
  {
    System.out.print("Do you wish to encrypt or decrypt? (e/d)  ");
    BufferedReader kbd = new BufferedReader(new InputStreamReader(System.in));
    char choice = kbd.readLine().charAt(0);

    // Let's use the Caesar cipher as our encryption scheme.  If we want
    // to change to another scheme, just use a different class that
    // implements the Encryption interface.
    Xor cipher = new Xor();

    if (choice == 'e')
    {
      System.out.print("Enter text to encrypt:  ");
      String input = kbd.readLine();
      cipher.setPlaintext(input);
      cipher.encrypt();
    }
    else if (choice == 'd')
    {
      System.out.print("Enter encrypted text:  ");
      String input = kbd.readLine();
      cipher.setCiphertext(input);
      cipher.decrypt();
    }
  }
}
```

```java
/** TwoD.java -- interface for 2-dimensional objects
 */
public interface TwoD
{
  public double findArea();
}
```

```java
/** Rectangle.java -- example of a 2-d shape
 *  We say that we are extending the Shape class, where the name attribute
 *  was declared.
 */
public class Rectangle extends Shape implements TwoD
{
  private double length;
  private double width;

  /** Notice that the first thing we do is call super(), which
   *  means we are calling our parent's constructor.
   */
  public Rectangle(String n, double l, double w)
  {
    super(n);
    length = l;
    width = w;
  }

  public double findArea()
  {
    return length * width;
  }
}
```

```java
/** Shape.java -- The most general class in our hierarchy.
 */
public class Shape
{
  private String name;

  public Shape(String n)
  {
    name = n;
  }

  public String toString()
  {
    return name;
  }
}
```

```java
/** ThreeD.java -- in addition to area, we want the ability to determine
 *  the volume.
 */
public interface ThreeD exte[illegible] TwoD
{
  public double findVolume();
}
```

```java
/** Sphere.java -- an example of a class implementing ThreeD.  This means
 *  we must specify both area and volume.
 */
public class Sphere extends Shape implements ThreeD
{
  private double radius;

  public Sphere(String n, double r)
  {
    super(n);
    radius = r;
  }

  // It might be faster to just use radius*radius, rather than
  // calling the pow() function.
  public double findArea()
  {
    return 4.0 * Math.PI * Ma[illegible]ow(radi[illegible] 2.0);
  }

  public double findVolume()
  {
    return 4.0/3.0 * Math.PI * Math.pow(radius, 3.0);
  }
}
```

```java
/** Cube.java -- another example of a class implementing ThreeD.
 */
public class Cube extends Shape implements ThreeD
{
  private double side;

  public Cube(String n, double s)
  {
    super(n);
    side = s;
  }

  public double findArea()
  {
    return 6.0 * Math.pow(side, 2.0);
  }

  public double findVolume()
  {
    return Math.pow(side, 3.0);
  }
}
```

```java
/** Driver.java -- Let's practice using objects of inherited classes.
 *  For example, both rectangles and spheres are specific kinds of shapes.
 *
 *  One interesting feature of inheritance is that when we declare our
 *  shape objects s1 and s2 below (not terrific names for variables!)
 *  we could have declared them to be of type Shape.  However, to call the
 *  proper findArea or findVolume functions, we would need to use a cast
 *  like this:  ((Rectangle) s1).findArea().
 */
public class Driver
{
  public static void main(String [] args)
  {
    Rectangle s1 = new Rectangle("office", 14, 18);
    Sphere s2 = new Sphere("volleyball", 4.5);

    System.out.println("The " + s1 + " has an area of " + s1.findArea());
    System.out.println("The " + s2 + " has a volume of " + s2.findVolume());
  }
}
```

Wow – we can have both inheritance & interfaces.

(We can even inherit from a parent interface!)

```java
// Here we define all the attributes and operations that every animal will
// have.  Other specialized classes for different types of animals will
// build on this basic definition.
// The classifier "protected" means that these attributes can be accessed
// in this file, and in subclasses, but nowhere else.
//
// By the way, I've included print statements in the constructors just so
// we can see what's going on.  In the long run, we usually wouldn't do this.
public class Animal
{
  protected double weight;
  protected boolean isWarmBlooded;
  protected boolean isHungry;

  public Animal()
  {
    weight = 10;
    isWarmBlooded = false;
    isHungry = true;
    System.out.println("Just created an animal.");
  }

  public String toString()
  {
    return "animal";
  }

  // An animal should only exercise if it's had enough to eat (if not hungry).
  // We also have a special cases for fish and birds.
  // Here in the Animal class we can look down and see if an Animal object is
  // a Fish or a Bird, but we can't call any specific Fish or Bird function.
  // So we can tell if we are a bird, but we can't tell if we can fly (oops).
  // We could write our own exercise() version in the Bird class.
  public void exercise()
  {
    if (isHungry)
      System.out.println("Too hungry to exercise, sorry.");
    else
    {
      if (this instanceof Fish)
        System.out.println(this + " swims about!");
      else if (this instanceof Bird)
          System.out.println("watch " + this + " fly!");
      else
        System.out.println(this + " gets a workout!");

      isHungry = true;
    }
  }

  // Only feed the animals if they are really hungry.
  public void feed()
  {
    if (! isHungry)
      System.out.println(this + " isn't hungry at the moment.  Try later.");
    else
    {
      System.out.println(this + " is grateful for the grub!");
      weight += 0.1;
      isHungry = false;
    }
  }

  public double getWeight()
  {
    return weight;
  }
}
```

```java
public class Fish extends Animal
{
  double length;
  boolean isSaltWater;

  public String toString()
  {
    return "Frank";
  }
}
```

```java
// Here we define what it means to be a bird...  We're making the attributes
// protected so that a subclass (e.g. Penguin) can change them.
public class Bird extends Animal
{
  protected double wingspan;

  // we also inherit the attributes:  isHungry, isWarmBlooded

  public Bird()
  {
    weight = 20;
    isWarmBlooded = true;
    wingspan = 30;
    System.out.println(" just created a bird");
  }

  public boolean canFly()
  {
    return true;
  }
}
```

```java
public class Penguin extends Bird
{
  // no new attributes -- just use the ones from Animal & Bird

  public Penguin()
  {
    System.out.println("  just created a penguin");
  }

  public boolean canFly()
  {
    return false;
  }

  public String toString()
  {
    return "Tux";
  }
}
```

```java
public class Reptile extends Animal
{
  public Reptile()
  {
    weight = 40;
  }

  public String toString()
  {
    return "reptile";
  }

  // When the Snake class calls super.getName(), this is it.
  public String getName()
  {
    return "Sam";
  }
}
```

```java
public class Snake extends Reptile
{
  private double length;
  private boolean isVenomous;

  public Snake()
  {
    length = 4;
    weight = 30;
    isVenomous = false;
  }

  // If we are in a subclass, it's possible to call a function
  // defined in the parent class by [illegible]g the w[illegible] "super".
  public String toString()
  {
    return super.getName();
  }
}
```

```java
/** Let's make a zoo -- the other files in this program define various
 *  animals, and here we can create and play with them.
 *  Because a zoo has several animals, and we don't know how many in advance,
 *  let's use a Vector instead of an array of animals.  A Vector has all the
 *  functionality and is more flexible than an array:  it allows you to grow
 *  or shrink whenever needed.
 */
import java.util.*;              // where the Vector class is defined
import java.io.*;

public class Driver
{
  public static void main(String [] args) throws IOException
  {
    Vector zoo = new Vector();
    BufferedReader kbd = new BufferedReader(new InputStreamReader(System.in));

    // This loop will interactively get input concerning
    // what the user wants to do with the animals.
    while (true)
    {
      System.out.println("\nCurrently we have " + zoo.size() + " animals.");
      System.out.print("(p)rint zoo, (a)dd animal, (d)elete, ");
      System.out.println("(e)xercise, (f)eed, (w)eigh, (q)uit program");
      System.out.print("Enter your selection:  ");
      char input = kbd.readLine().charAt(0);

      // A series of if/else statements will handle the individual commands.
      // It would be more efficient to read both the command and the
      // respective animal at the same time, rather than have separate prompts,
      // but it would be more difficult for the user to remember exactly
      // how to enter the commands.  This implementation makes it easy to run.
      if (input == 'a')
      {
        System.out.print("(b)ird, (p)enguin, (f)ish, (r)eptile, (s)nake?  ");
        char type = kbd.readLine().charAt(0);
        switch(type)
        {
          case 'b' : zoo.add(new Bird()); break;
          case 'f' : zoo.add(new Fish()); break;
          case 'p' : zoo.add(new Penguin()); break;
          case 'r' : zoo.add(new Reptile()); break;
          case 's' : zoo.add(new Snake()); break;
          default: System.out.println("Invalid animal type, sorry.");
        }
      }
      else if (input == 'd')
      {
        System.out.print("Enter number of animal to delete:  ");
        int num = Integer.parseInt(kbd.readLine());
        zoo.remove(num);
      }
      else if (input == 'p')
      {
        for (int i = 0; i < zoo.size(); ++i)
          System.out.println("#" + i + " --> " + zoo.elementAt(i));
      }

      // Take a close look at the calls to exercise() and feed() below.
      // It turns out that the Vector's elementAt() function returns simply
      // an Object, and we need to cast it to Animal, because exercise()
      // and feed() belong to the Animal class.  This cast is not a problem,
      // because the Vector consists of animal objects (or specialized
      // versions of animals, which also have the ability to exercise and eat)
      else if (input == 'e')
      {
        System.out.print("Which animal do you want to exercise?  ");
        int num = Integer.parseInt(kbd.readLine());
        ((Animal)(zoo.elementAt(num))).exercise();
      }
      else if (input == 'f')
      {
        System.out.print("Which animal do you want to feed?  ");
        int num = Integer.parseInt(kbd.readLine());
        ((Animal)(zoo.elementAt(num))).feed();
      }
      else if (input == 'w')
      {
        System.out.print("Which animal do you want to weigh?  ");
        int num = Integer.parseInt(kbd.readLine());
        System.out.println(zoo.elementAt(num) + " weighs " +
                           ((Animal)(zoo.elementAt(num))).getWeight());
      }
      else if (input == 'q')
        break;
      else
        System.out.println("Invalid option.  Try again.");
    }
  }
}
```