- **Abstract Class**

A superclass defines common operations for the subclasses. In the inheritance hierarchy, classes become more specific and concrete with each level of new subclasses. If you move from a subclass back up to a superclass, the class becomes more general and less specific. Class design should ensure a superclass contains common operations of its subclasses.

There are special cases that a superclass does not have the full implementation of the class methods since the behaviors are specific to the subclasses. For example, the formula for computing the area of a rectangle and the area of a circle are different. Although the getArea() method is defined in both Rectangle class and Circle class, it is a good practice to define the method in the Shape class because it is a common feature that all Shape objects will have. However, because how the area is computed is different among different Shape objects, the getArea() method can be left unimplemented and defined as an "abstract method" in the Shape class. A class containing abstract methods is called an **abstract class**. In other words, an abstract class is a class with one or more abstract methods, which are the methods with no body, not even an empty one.

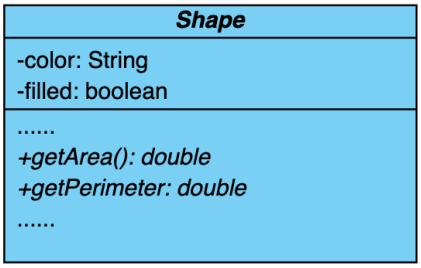
Defining abstract methods in a superclass can be considered as a contract between the superclass and the subclasses. The classes that extend an abstract class must implement the abstract methods. An abstract class can be used as a data type to declare a variable; however, it cannot be used to instantiate an object (**new** an object in Java) since it does not have the full implementation. An abstract class can contain abstract methods that must be implemented in subclasses, which are also called **concrete subclasses**. In Java, the "abstract" keyword must be used in the class header if a class contains abstract methods. The "abstract" keyword must be used in an abstract method signature as well. An example is shown below.

```
public abstract class Figure {
   private int fixedX, fixedY;
   public Figure( int fx, int fy ) {
   ......
   }
   public abstract void draw();   // No way to draw it!
}

public class Rectangle extends Figure {
//must implement draw(), or abstract keyword is needed
   private int width, height;
   public Rectangle ( ...... ) {
   }
   public void draw() {
      ......  // the concrete method to draw the rectangle
   }
}
```

A class that contains abstract methods must be abstract. However, it is possible to define an abstract class that doesn't contain any abstract methods. This abstract class is used as a base class for defining subclasses. A subclass can be abstract even if its superclass is concrete. For example, the Object class is concrete, but its subclasses, such as the Shape class, may be abstract.

In a class diagram, the names of abstract classes and their abstract methods are *italicized*. For example, in the class diagram below, the name of the class "Shape" is italicized, as well as the names of the abstract methods "getArea()" and "getPerimeter()".



An abstract class can be used as a regular superclass as a data type, but remember you **cannot create an instances of an abstract class** using the **new** operator. For example, the following statement, which creates an array of Shape objects is valid and does not cause an error at the compile time.

```
Shape[] shapes = new Shape[10];
```

The above statement allocates a block of memory addresses for processing an array of shape objects, which might include a mix of circles and rectangles. Next, you can create an instance of the subclasses, for example the Circle class, and store the reference in the array like this:

```
shapes[0] = new Circle();
```

On the other hand, the following statement will cause a compilation error since you are trying to create an instance of the abstract class "Shape".

```
Shape s = new Shape();
```

The constructor in the abstract class can be defined as **protected** because it is used only by the subclasses. Use the "super" keyword in the subclasses' constructors to invoke the constructors and initialize the data fields defined in the abstract classes.

- **Interfaces**

Java doesn't allow multiple inheritance (while C++ does) since multiple inheritance is problematic. However, Java has another type of inheritance: **Interface Inheritance**. Adding interface inheritance gives a lot of the power of multiple inheritance, but none of the problems. A Java "Interface" is a special "class" with no data and only abstract methods. A class can "extend" only one class but can implement many interfaces. For example,

```
public class AAA extends BBB implements CCC, DDD, EEE {
   .....
}
```

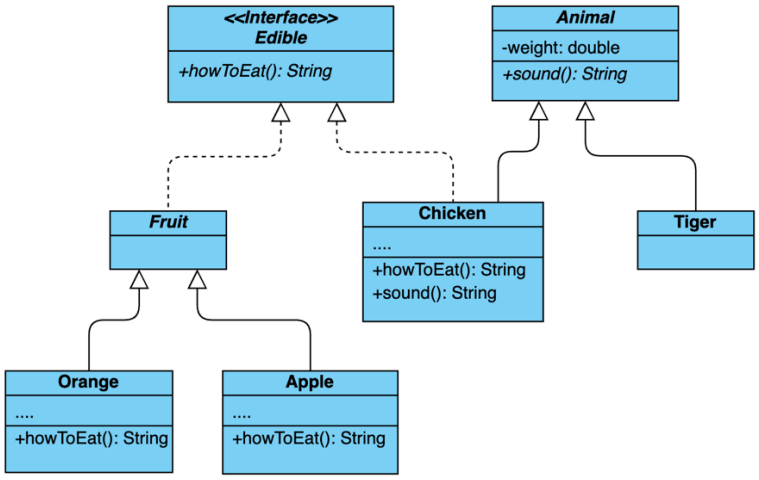
A Java **Interface** defines common operations for objects. In many ways, an interface is similar to an abstract class, but its intent is to specify common operations for the objects of related classes or unrelated classes. For example, using appropriate interfaces, you can specify that the objects are comparable, edible, and/or cloneable. Implementing an interface allows a class to become more formal about the behavior it promises to provide. A Java Interface form a contract between the class and the outside world, and this contract is enforced at build time by the compiler. Java has lots of Interfaces, one of them is **Comparable,** which has one method as shown below that works somewhat like strcmp in C++: returns a negative integer if less than X, returns 0 if equals X, returns a positive integer greater than X.

```
public interface Comparable {
public int compareTo ( Object x );
}

public class Date implements Comparable {
   ...
   public int compareTo ( Object x ) {
      // check years, if equal, then months, if equal, then days
   }
}
```

A Java Interface is treated like a special class. The keyword "interface" is used to define a Java Interface. Each interface is compiled into a separate bytecode file, just like a regular class. You can use an interface more or less the same way you use an abstract class. For example, you can use an interface as a data type for a reference variable, as the result of casting, and so on. Just like an abstract class, an interface cannot be used to create an instance of the interface using the **new** operator.

As an example, in the following class diagram, you can use the **Edible interface** to specify whether an object is edible. This is accomplished by letting the class for the object implements this interface using the **implements** keyword in Java code. Note that, in a class diagram, you use the stereotype **<<interface>>** above the class name to represent an interface class. The name of the interface is italicized. An arrow with a dashed line for the arrow shaft indicates implementation of an interface. In UML, such a relationship is called a "realization".



In the class diagram above, the Chicken class and Fruit class implement the Edible interface. The relationship between the class and the interface is known as interface inheritance. Edible interface is a supertype for Chicken class and Fruit class. Animal class is a supertype for Chicken class and Tiger class. Fruit class is a supertype for Orange class and Apple class. The Animal class defines the weight property and sound method. The sound method is an abstract method and will be implemented by a concrete animal class. The Chicken class implements Edible interface to specify that chicken objects are edible. When a class implements an interface, it must implement all the methods defined in the interface. The Chicken class must implement the howToEat() method. Chicken class also extends the Animal class and must implement the sound() method. If the Fruit class implements the Edible interface but does not implement the howToEat() method, Fruit class must be defined as abstract. The concrete subclasses of

the Fruit class, Orange class and Apple class, must implement the howToEat() method. In essence, the Edible interface defines common operations for edible objects. All edible objects have the howToEat() method.

As another example, the code below defines a Bicycle interface. The ACMEBicycle implements the Bicycle interface and promises to have the concrete behaviors implemented as defined in the Bicycle interface, including four abstract methods.

```
public interface Bicycle {
//  wheel revolutions per minute
   void changeCadence(int newValue);
   void changeGear(int newValue);
   void speedUp(int increment);
   void applyBrakes(int decrement);
}
```

```
public class ACMEBicycle implements Bicycle {
   private int cadence = 0;
   private int speed = 0;
   private int gear = 1;
// The compiler will now require that methods
// changeCadence, changeGear, speedUp, and applyBrakes
// all be implemented. Compilation will fail if those
// methods are missing from this class.
   void changeCadence(int newValue) {
      cadence = newValue;
   }
   void changeGear(int newValue) {
      gear = newValue;
   }
   void speedUp(int increment) {
      speed = speed + increment;
   }
   void applyBrakes(int decrement) {
      speed = speed - decrement;
   }
   void printStates() {
      System.out.println("cadence:" + cadence + " speed:"
             + speed + " gear:" + gear);
   }
}
```

The modifiers **public static final** on the data fields and the modifiers **public abstract** on methods in a Java interface can be omitted. Therefore, the following interface definitions are equivalent:

```
public interface T {
    public static final int k = 1;
    public abstract void p();
}
```

is equivalent to

```
public interface T {
    int k = 1;
    void p();
}
```

Although the **public** modifier may be omitted for a method defined in a Java interface, the method must be defined as **public** when it is implemented in a subclass.

- **Example 1**

  Suppose A is an abstract class, B is a concrete subclass of A, and both A and B have a default constructor. Which of the following is correct?

```
(a)   A a = new A();
(b)   A a = new B();
(c)   B b = new A();
(d)   B b = new B();
```

- **Example 2**

  Show the output of running the class Example in the following code lines:

```
interface A {
}

class C {
}

class D extends C {
}

class B extends D implements A {
}

public class Example {
    public static void main(String[] args) {
        B b = new B();
        if (b instanceof A)
            System.out.println("b is an instance of A");
        if (b instanceof C)
            System.out.println("b is an instance of C");
    }
}
```

- **Example 3**

What is the sequence of constructor chaining and what is the output of running class Example3?

```
public class Example3 {
   public static void main(String[] args) {
         new Circle9();
   }
}

public abstract class GeometricObject {
   protected GeometricObject() {
         System.out.print("A");
   }

   protected GeometricObject(String color, boolean filled) {
        System.out.print("B");
   }
}

public class Circle9 extends GeometricObject {
    public Circle9() {
        this(1.0);
        System.out.print("C");
    }

    public Circle9(double radius) {
        this(radius, "white", false);
        System.out.print("D");
    }

    public Circle9(double radius, String color, boolean filled) {
        super(color, filled);
        System.out.print("E");
    }
}
```