- **Abstract Classes**

A superclass defines common behavior for related subclasses. An interface can be used to define common behavior for classes (including unrelated classes.) An interface is for defining common behavior for classes (including unrelated classes). Before discussing interfaces, let's discuss a closely related subject: abstract classes.

An abstract class cannot be used to instantiate objects. An abstract class can contain abstract methods that must be implemented in subclasses, which are also called concrete 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 features of its subclasses. Sometimes, a superclass is so abstract that it cannot implement the specific behaviors to the subclasses, thus it cannot be used to create any specific instances. Such a class is referred to as 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. For example,

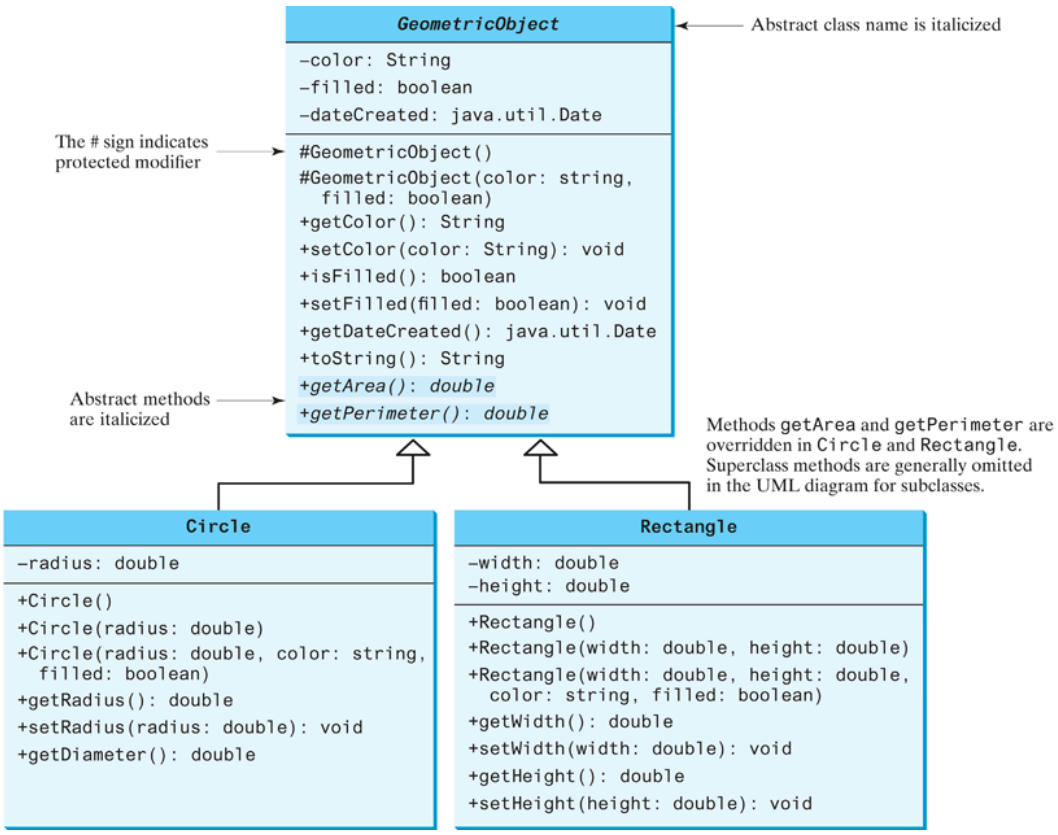
```
public abstract class Figure {
   private int fixedX, fixedY;
   public Figure( int fx, int fy ) {
   ......
   }
   public abstract void draw();   // No way to draw it!
}

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 GeometricObject, may be abstract.

In the previous lecture notes, GeometricObject was defined as the superclass for Circle and Rectangle. GeometricObject models common features of geometric objects. Both Circle and Rectangle contain the getArea() and getPerimeter() methods for computing the area and perimeter of a circle and a rectangle. Since you can compute areas and perimeters for all geometric objects, it is better to define the getArea() and getPerimeter() methods in the GeometricObject class. However, these methods cannot be implemented in the GeometricObject class because their implementation depends on the specific type of geometric object. Such methods are referred to as abstract methods and are denoted using the **abstract modifier** in the method header. After you define the methods in GeometricObject, it becomes an abstract

class. Abstract classes are denoted using the abstract modifier in the class header. In UML graphic notation, the names of abstract classes and their abstract methods are *italicized*, as shown below.



Abstract classes are like regular classes, but you **cannot create instances of abstract classes** using the **new** operator. However, an abstract class can be used as a data type. Therefore, the following statement, which creates an array whose elements are of the GeometricObject type, is valid:

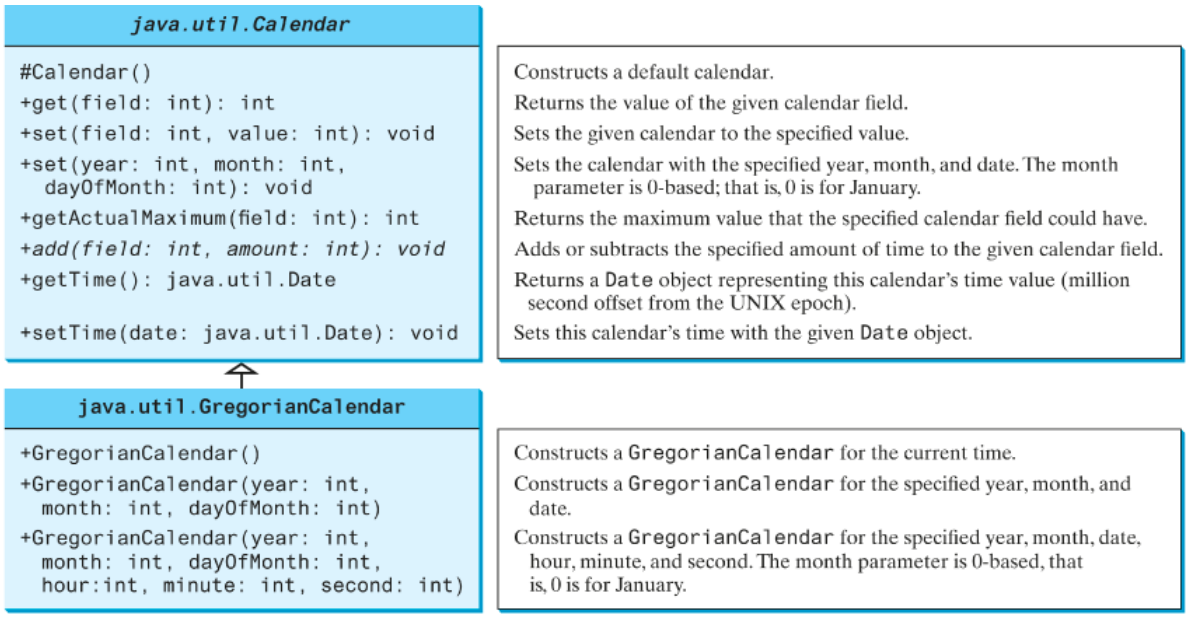
GeometricObject[] objects = new GeometricObject[10];

You can then create an instance of GeometricObject and assign its reference to the array like this:

objects[0] = new Circle();

An abstract method is defined without implementation. Its implementation is provided by the subclasses. A class that contains abstract methods must be defined as abstract. 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 superclass.

Another example.



- **Interfaces**

Java doesn't allow multiple inheritance (C++ does). Some "problems" of multiple inheritance: if B extends A, C extends A, what if D extends B, C? 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.

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

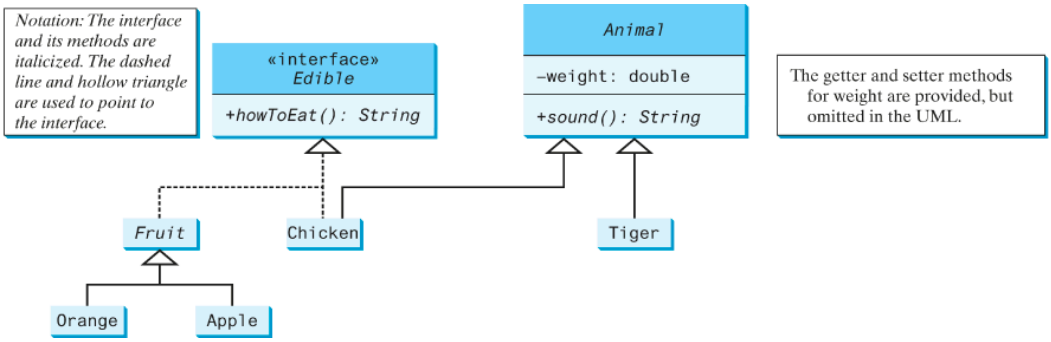
An interface is a special "class" with no data and only abstract methods. In general, a class can "extend" only one class but implement many interfaces. An interface is a class-like construct for defining common operations for objects. In many ways an interface is similar to an abstract class, but its intent is to specify common behavior for 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 < 0 if less than X, = 0 if equals X, > 0 if 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
   }
}
```

An interface is treated like a special class in Java. 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. As with an abstract class, you cannot create an instance from an interface using the new operator. You can use the Edible interface to specify whether an object is edible. This is accomplished by letting the class for the object implement this interface using the implements keyword. For example, the classes Chicken and Fruit implement the Edible interface as shown in the Class Diagram below. The relationship between the class and the interface is known as interface inheritance. Since interface inheritance and class inheritance are essentially the same, we will simply refer to both as inheritance.



Edible is a supertype for Chicken and Fruit. Animal is a supertype for Chicken and Tiger. Fruit is a supertype for Orange and Apple. 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 to specify that chickens are edible. When a class implements an interface, it implements all the methods defined in the interface. The Chicken class must implement the howToEat method. Chicken also extends Animal to implement the sound method. If the Fruit class implements but does not implement the howToEat method, Fruit class must be defined as abstract. The concrete subclasses of Fruit must implement the howToEat method. In essence, the Edible interface defines common behavior for edible objects. All edible objects have the howToEat method.

As another example, the ACMEBicycle implements the Bicycle interface and promises to have concrete behaviors implemented as defined in the ACMEBicycle interface.

```
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 data fields and the modifiers public abstract on methods can be omitted in an interface. Therefore, the following interface definitions are equivalent:

```
public interface T {
  public static final int K = 1;

  public abstract void p();
}
```

Equivalent

```
public interface T {
  int K = 1;

  void p();
}
```

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