COP 2250 – Chapter 13 – Abstract Classes and Interfaces

Abstract Classes

* Abstract classes are classes that will never be instantiated, only subclassed.
* They exist only to be used as superclasses.
* Abstract classes are common in the libraries of the Java API .
* Abstract classes are used when a superclass is needed for a number of subclasses, but there will never be a need for the superclass instance.
* Abstract methods in the abstract superclass are implemented in the subclasses for some useful purposes, such as comparing areas in the GeometricObject example (see next).

Another GeometricObject Example

* There is a new, abstract version of class GeometricObject on pages 500-502. Note that there are two abstract methods at the end of this class definition.
* Both of these abstract methods are overridden in the subclasses Circle and Rectangle, so these latter two classes don’t have to be abstract.
* TestGeometricObject on page 502-503 tests the example.

Try TestGeometricObject

Interesting Point about abstract Classes (See also bullet list on page 504)

1. Abstract classes are declared with the keyword **abstract.**

**public abstract class WhatEver {**

1. Abstract classes cannot be instantiated with new, only extended.
2. They might (not a requirement) contain abstract methods and non-abstract methods.
3. Abstract methods are declared with the abstract keyword in the method header.

**public abstract double doSomething(); // NOTE: no body, ends with ;**

1. Abstract methods are not implemented in the abstract class. There is no body block and no curly braces.
2. Abstract methods must be implemented in a subclass (with a body block). If not, the subclass must also be declared as abstract.
3. Abstract methods cannot be static.
4. A class containing even one abstract method must be declared as an abstract class.
5. An abstract class can be a subclass of a non-abstract, or concrete, superclass.
6. A non-abstract method in a superclass can be overridden as abstract in a subclass.
7. You can use an abstract class as a data type.

Case Study: The Abstract Number Class

* As shown in Figure 13.2, Number is the superclass for the numeric wrapper classes.

Try LargestNumbers

Case Study: Calendar and GregorianCalendar

* Study the UML in Figure 13.3 and the Calendar class constant in Table 13.1

Try TestCalendar

**Exercise**: Write a program to determine what day Christmas falls on in 2018.

Interfaces

* An **interface** in Java is a “class-like” structure that classes can implement.
* Interfaces are intended for objects to exhibit common behaviors for comparisons.
* Interfaces may contain only **abstract method signatures** and **constants**.
* Interfaces are declared using the **interface** keyword.

**public interface Edible {**

**public abstract void howToEat();**

**}**

* An interface may never contain method definitions.
* As interfaces are implicitly abstract, they cannot be directly instantiated with the new operator. They must be implemented by a class with the **implements** keyword.

**public class Banana implements Edible {**

**public void howToEat() {**

**System.out.println(“Peel. Open mouth and move towards face”);**

**System.out.println(“In peanut butter sandwiches”); //uh huhuh**

**}**

**}**

Try TestGeometricObject

Study the Edible interface on page 510.

* Examine TestEditable on pages 510-511, noting internal classes that implement Edible.
* Study Figure 13.4 to understand the inheritance hierarchy in this example.

Try TestEdible

* The implementing class must implement all of the methods described in the interface, or be an abstract class. Eclipse helps with this requirement.
* Since multiple inheritance of classes is not allowed in Java, one benefit of using interfaces is that they simulate multiple inheritance. A Java class/interface may implement/extend any number of interfaces.
* Interfaces are used to encode similarities which classes of various types share, but do not necessarily constitute a class relationship. For instance, a human and a parrot can both whistle. However, it would not make sense to represent Humans and Parrots as subclasses of a Whistler class. Instead, Humans and Parrots would most likely be subclasses of an Animal class, but both would implement the Whistler interface.
* The body of the interface contains abstract methods, but since all methods in an interface are, by definition abstract, the abstract keyword is not required. Since all methods are implicitly public, the public keyword can also be omitted.
* If a class is abstract, one of its subclasses is expected to implement its unimplemented methods.
* Interfaces can **extend** several other interfaces.

The Comparable Interface

* The API’s Comparable interface has the method **compareTo()** which is used to describe two objects as equal, or to indicate one is greater than or less than, the other.
* Comparable is a generic interface, as evidenced by the generic notation **<E>**. See page 514.
* The E will be replaced by a specific object type when the interface is implemented.
* The compareTo(E ob) method compares the two objects and returns:
  + A positive integer if the object implementing Comparable is greater than ob.
  + A negative integer if the object implementing Comparable is less than ob.
  + Zero if the object implementing Comparable is equal to ob.
* To see an example of coding compareTo(), see lines 9-16 in ComparableRectangle on page 515.
* Many classes in the API were coded to implement Comparable to enable sorting. See page 514.

Try SortComparableObjects

* The ComparableRectangle class on page 511 is coded to implement Comparable. Study it.

Try SortRectangles

The Cloneable Interface

* This interface is used to create objects that can be exactly copied, or cloned,
* The Cloneable interface is surprisingly, **empty**! It’s a marker interface. See page 518.
* The **clone()** method of the **Object** class is used to clone an object that implements Cloneable.
* Many API classes can be cloned such as ArrayList, Date, and Calendar.
* Arrays can be cloned because the API defines an array to implement Cloneable.
* A custom class that implements Cloneable must override the Object class clone() method and change its modifier from protected to public. See page 519.
* If not, a **CloneNotSupportedException** will be thrown.
* Examine House on pages 514-515.

Exercise

Liang didn’t provide a test program for House. Write a main class named **TestHouse** that does.

Interfaces vs. Abstract Classes

* See Table 13.2 to compare and contrast these constructs.
* Note that an interface can extend any number of other interfaces, but not classes.
* A class can extend one class and implement any number of interfaces.
* All classes extend Object, but there is no “root” for interfaces.
* For clarity, use nouns for class names and adjectives or nouns for interfaces.
* Use abstract classes when a strong **is-a relationship** is present.
* Use interfaces when there is a weak is-a relationship (an **is-kind-of relationship**).
* Interfaces generally are more flexible.

Case Study: The Rational Class

* See page 528 for a custom (not API) class for rational numbers named Rational.

Exercise

* Liang doesn’t provide a test class for Rational. Make one named TestRationalClass.
* See page 530 for suggested (although flawed) code.

Class Design Guidelines

* See pages 531-533 for tips and suggestions.