Abstract Classes and Interfaces

CSE 114: Introduction to Object-Oriented Programming
Paul Fodor
Stony Brook University
http://www.cs.stonybrook.edu/~cse114
Contents

- Abstract Classes and Abstract Methods
  - abstract methods and constraints on the subclasses of abstract classes
  - abstract classes as types
  - The abstract Calendar class and its GregorianCalendar subclass

- Interfaces:
  - Defining and Implementing Interfaces
  - The Comparable Interface
    - Writing a generic max Method
    - Defining Classes to Implement Comparable
  - The Cloneable Interface
    - Implementing the Cloneable Interface
    - Shallow vs. Deep Copy
  - Interfaces vs. Abstract Classes
  - Interface Inheritance
  - Conflicting interfaces
Abstract Classes and Abstract Methods

Abstract classes are italicized or have the annotation <abstract>.

Abstract class constructors have protected modifier.

Abstract methods are italicized or have the annotation <abstract>.

Methods getArea and getPerimeter are overridden in Circle and Rectangle. Superclass abstract methods are generally omitted in the UML diagram for subclasses.
public abstract class GeometricObject {
    private String color = "white";
    private boolean filled;
    private java.util.Date dateCreated;
    protected GeometricObject() {
        dateCreated = new java.util.Date();
    }
    protected GeometricObject(String color, boolean filled) {
        dateCreated = new java.util.Date();
        this.color = color;
        this.filled = filled;
    }
    public String getColor() {   return color;  }
    public void setColor(String color) {  this.color = color;  }
    public boolean isFilled() {   return filled;  }
    public void setFilled(boolean filled) {  this.filled = filled;  }
    public java.util.Date getDateCreated() {   return dateCreated;  }
    public String toString() {
        return "created on " + dateCreated + "\n            color: " + color + " and filled: " + filled;
    }
    /** Abstract method getArea */
    public abstract double getArea();
    /** Abstract method getPerimeter */
    public abstract double getPerimeter();
}
public class Circle extends GeometricObject {
    private double radius;
    public Circle() {
    }
    public Circle(double radius) {
        this.radius = radius;
    }
    public double getRadius() {
        return radius;
    }
    public void setRadius(double radius) {
        this.radius = radius;
    }
    public double getArea() {
        return radius * radius * Math.PI;
    }
    public double getPerimeter() {
        return 2 * radius * Math.PI;
    }
    public double getDiameter() {
        return 2 * radius;
    }
}
public class Rectangle extends GeometricObject {
    private double width;
    private double height;
    public Rectangle() {
        // super();
    }
    public Rectangle(double width, double height) {
        this();
        this.width = width;
        this.height = height;
    }
    public Rectangle(double width, double height, String color,
            boolean filled) {
        super(color,filled);
        this.width = width;
        this.height = height;
    }
    public double getWidth() {    return width;  }
    public void setWidth(double width) {
        this.width = width;  }
    public double getHeight() {    return height;  }
    public void setHeight(double height) {
        this.height = height;  }
    public double getArea() {
        return width * height;
    }
    public double getPerimeter() {
        return 2 * (width + height);
    }
}

(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)
public class TestGeometricObject1 {
    public static void main(String[] args) {
        // Declare and initialize two geometric objects
        GeometricObject geoObject1 = new Circle(5);
        GeometricObject geoObject2 = new Rectangle(5, 3);
        // Display circle
        displayGeometricObject(geoObject1);
        // Display rectangle
        displayGeometricObject(geoObject2);
        System.out.println("The two objects have the same area? " +
                equalArea(geoObject1, geoObject2));
    }

    /** A method for displaying a geometric object */
    public static void displayGeometricObject(GeometricObject object) {
        System.out.println(object); // object.toString()
        System.out.println("The area is " + object.getArea());
        System.out.println("The perimeter is " + object.getPerimeter());
    }

    /** A method for comparing the areas of two geometric objects */
    public static boolean equalArea(GeometricObject object1,
                            GeometricObject object2) {
        return object1.getArea() == object2.getArea();
    }
}

(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)
abstract methods in abstract classes

• An abstract method can only be contained in an abstract class.
subclasses of abstract classes

- In a nonabstract (also called concrete) subclass extended from an abstract super-class, all the abstract methods MUST be implemented.

```java
abstract class A {
    abstract void m();
}

class B extends A {
    void m() {
    }
}
```
subclasses of abstract classes

- In an abstract subclass extended from an abstract super-class, we can choose:
  - to implement the inherited abstract methods OR
  - to postpone the constraint to implement the abstract methods to its nonabstract subclasses.
abstract class A {
    abstract void m();
}

abstract class B extends A {
    void m() {
    }
}

class C extends B {
}

(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)
abstract class A{
    abstract void m();
}

abstract class B extends A{
}

public class C extends B {
    void m()
    {
    }
}

(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)
abstract classes

• A subclass can be abstract even if its superclass is concrete.

• For example, the `Object` class is concrete, but a subclass, `GeometricObject`, is abstract.
**abstract classes**

- A subclass can override a method from its concrete superclass to define it **abstract**
- useful when we want to **force** its subclasses to implement that method, or
- the implementation of the method in the superclass is invalid in the subclass
abstract classes

• It is possible to define an abstract class that contains no abstract methods.
• This class is used as a base class for defining new subclasses.
abstract classes

• An object **cannot** be created from abstract class:
  • An abstract class cannot be instantiated using the `new` operator:
    ```java
    GeometricObject o = new GeometricObject();
    ```

• We still define its constructors, which are invoked in the constructors of its subclasses through **constructor chaining**.
  • For instance, the constructors of `GeometricObject` are invoked by the constructors in the `Circle` and the `Rectangle` classes.
abstract classes as types

• An abstract class can be used as a data type:
  GeometricObject c = new Circle(2);
• We can also create an array whose elements are of the abstract GeometricObject type:
  GeometricObject[] geo = new GeometricObject[10];
• There are only null reference elements in the array until they are initialized with concrete objects:
  geo[0] = new Circle();
  geo[1] = new Rectangle();
  ...

The abstract Calendar class and its GregorianCalendar subclass

- An instance of `java.util.Date` represents a specific instant in time with millisecond precision.

- `java.util.Calendar` is an abstract base class for extracting detailed information such as `year`, `month`, `date`, `hour`, `minute` and `second` from a `Date` object for a specific calendar.

- Subclasses of `Calendar` can implement specific calendar systems such as Gregorian calendar, Lunar Calendar and Jewish calendar.

- `java.util.GregorianCalendar` is for the modern Gregorian calendar.
The GregorianCalendar Class

- Java API for the `GregorianCalendar` class:
  [http://docs.oracle.com/javase/8/docs/api/java/util/GregorianCalendar.html](http://docs.oracle.com/javase/8/docs/api/java/util/GregorianCalendar.html)
- `new GregorianCalendar()` constructs a default `GregorianCalendar` with the current time
- `new GregorianCalendar(year, month, date)` constructs a `GregorianCalendar` with the specified `year`, `month`, and `date`
  - The `month` parameter is 0-based, i.e., 0 is for January, 1 is for February, ..., 11 is for December.
The abstract Calendar class and its GregorianCalendar subclass

**java.util.Calendar**

- Calendar()
  - Constructs a default calendar.
- get(field: int): int
  - Returns the value of the given calendar field.
- set(field: int, value: int): void
  - Sets the given calendar to the specified value.
- set(year: int, month: int, dayOfMonth: int): void
  - Sets the calendar with the specified year, month, and date. The month parameter is 0-based, that is, 0 is for January.
- getActualMaximum(field: int): int
  - Returns the maximum value that the specified calendar field could have.
- add(field: int, amount: int): void
  - Adds or subtracts the specified amount of time to the given calendar field.
- getTime(): java.util.Date
  - Returns a Date object representing this calendar’s time value (million second offset from the Unix epoch).
- setTime(date: java.util.Date): void
  - Sets this calendar’s time with the given Date object.

**java.util.GregorianCalendar**

- GregorianCalendar()
  - Constructs a GregorianCalendar for the current time.
- GregorianCalendar(year: int, month: int, dayOfMonth: int)
  - Constructs a GregorianCalendar for the specified year, month, and day of month.
- GregorianCalendar(year: int, month: int, dayOfMonth: int, hour: int, minute: int, second: int)
  - Constructs a GregorianCalendar for the specified year, month, day of month, hour, minute, and second. The month parameter is 0-based, that is, 0 is for January.
The get Method in the Calendar Class

- The get(int field) method defined in the Calendar class is useful to extract the date and time information from a Calendar object.
- The fields are defined as constants in Calendar, as shown in the following:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>YEAR</td>
<td>The year of the calendar.</td>
</tr>
<tr>
<td>MONTH</td>
<td>The month of the calendar with 0 for January.</td>
</tr>
<tr>
<td>DATE</td>
<td>The day of the calendar.</td>
</tr>
<tr>
<td>HOUR</td>
<td>The hour of the calendar (12-hour notation).</td>
</tr>
<tr>
<td>HOUR_OF_DAY</td>
<td>The hour of the calendar (24-hour notation).</td>
</tr>
<tr>
<td>MINUTE</td>
<td>The minute of the calendar.</td>
</tr>
<tr>
<td>SECOND</td>
<td>The second of the calendar.</td>
</tr>
<tr>
<td>DAY_OF_WEEK</td>
<td>The day number within the week with 1 for Sunday.</td>
</tr>
<tr>
<td>DAY_OF_MONTH</td>
<td>Same as DATE.</td>
</tr>
<tr>
<td>DAY_OF_YEAR</td>
<td>The day number in the year with 1 for the first day of the year.</td>
</tr>
<tr>
<td>WEEK_OF_MONTH</td>
<td>The week number within the month.</td>
</tr>
<tr>
<td>WEEK_OF_YEAR</td>
<td>The week number within the year.</td>
</tr>
<tr>
<td>AM_PM</td>
<td>Indicator for AM or PM (0 for AM and 1 for PM).</td>
</tr>
</tbody>
</table>
import java.util.*;
public class TestCalendar {
    public static void main(String[] args) {
        // Construct a Gregorian calendar for the current date and time

        Calendar calendar = new GregorianCalendar();
        System.out.println("Current time is " + new Date());
        System.out.println("YEAR: \t" + calendar.get(Calendar.YEAR));
        System.out.println("MONTH: \t" + calendar.get(Calendar.MONTH));
        System.out.println("DATE: \t" + calendar.get(Calendar.DATE));
        System.out.println("HOUR: \t" + calendar.get(Calendar.HOUR));
        System.out.println("HOUR_OF_DAY: \t" + calendar.get(Calendar.HOUR_OF_DAY));
        System.out.println("MINUTE: \t" + calendar.get(Calendar.MINUTE));
        System.out.println("SECOND: \t" + calendar.get(Calendar.SECOND));
        System.out.println("DAY_OF_WEEK: \t" + calendar.get(Calendar.DAY_OF_WEEK));
        System.out.println("DAY_OF_MONTH: \t" + calendar.get(Calendar.DAY_OF_MONTH));
        System.out.println("DAY_OF_YEAR: " + calendar.get(Calendar.DAY_OF_YEAR));
        System.out.println("WEEK_OF_MONTH: " + calendar.get(Calendar.WEEK_OF_MONTH));
        System.out.println("WEEK_OF_YEAR: " + calendar.get(Calendar.WEEK_OF_YEAR));
        System.out.println("AM_PM: " + calendar.get(Calendar.AM_PM));
        // Construct a calendar for January 1, 2020
        Calendar calendar1 = new GregorianCalendar(2020, 0, 1);
        System.out.println("January 1, 2020 is a " + dayNameOfWeek(calendar1.get(Calendar.DAY_OF_WEEK)));
    }

    public static String dayNameOfWeek(int dayOfWeek) {
        switch (dayOfWeek) {
            case 1: return "Sunday";
            case 2: return "Monday";
            case 3: return "Tuesday";
            ... case 7: return "Saturday";
            default: return null;
        }
    }
}
Interfaces

• An **interface** is a class-like construct that contains only abstract methods and constants.

• Why is an interface useful?
  • An interface is similar to an abstract class, but the intent of an interface is to **specify behavior** for objects.
    • For example: specify that the objects are **comparable**, **edible**, **cloneable**, …
  • Allows multiple inheritance: a class can implement multiple interfaces.
Defining Interfaces

• Declaration:

```java
public interface InterfaceName {
   // constant declarations;
   // method signatures;
}
```
Implementing Interfaces

- The **Edible** interface specifies whether an object is edible
  
  ```java
  public interface Edible {
      public abstract String howToEat();
  }
  ```

- The class **Chicken** *implements* the **Edible** interface:

  ```java
  class Chicken extends Animal implements Edible {
      public String howToEat() {
          return "Chicken: Fry it";
      }
  }
  ```
interface Edible {
    public abstract String howToEat(); /** Describe how to eat */
}
abstract class Animal {
}
class Chicken extends Animal implements Edible {
    public String howToEat() {
        return "Chicken: Fry it";
    }
}
class Tiger extends Animal {
} /** Does not extend Edible */
abstract class Fruit implements Edible {
}
class Apple extends Fruit {
    public String howToEat() {
        return "Apple: Make apple cider";
    }
}
class Orange extends Fruit {
    public String howToEat() {
        return "Orange: Make orange juice";
    }
}

public class TestEdible {
    public static void main(String[] args) {
        Object[] objects = {new Tiger(), new Chicken(), new Apple()};
        for (int i = 0; i < objects.length; i++)
            if (objects[i] instanceof Edible)
                System.out.println(((Edible)objects[i]).howToEat());
    }
}
Omitting Modifiers in Interfaces

• In an interface:
  • All data fields are **public static final**
  • All methods are **public abstract**
  • These modifiers can be omitted:

```
public interface T1 {
  public static final int K = 1;
  public abstract void p();
}
```

Equivalent:
```
public interface T1 {
  int K = 1;
  void p();
}
```

• A constant defined in an interface can be accessed using `InterfaceName.CONSTANT_NAME`, for example: `T1.K`
Interfaces

• An interface is treated like a special class in Java:
  • Each interface is compiled into a separate bytecode file just like a regular class.
  • Like an abstract class, you cannot create an instance from an interface using the `new` operator.

• Uses of interfaces are like for abstract classes:
  • as a data type for a variable
  • as the result of casting
The Comparable Interface

- The **Comparable** interface is defined in the `java.lang` package and it is used by `Arrays.sort`

```java
package java.lang;
public interface Comparable {
    int compareTo(Object o);
}
```
Many classes in the Java library implement `Comparable` (e.g., `String` and `Date`) to define a natural order for the objects:

```java
public class String extends Object 
    implements Comparable { 
  // class body omitted 
}

public class Date extends Object 
    implements Comparable { 
  // class body omitted 
}
```

- `new String()` instanceof `String`  
- `new String()` instanceof `Comparable`  
- `new java.util.Date()` instanceof `java.util.Date`  
- `new java.util.Date()` instanceof `Comparable`
In UML, dashed lines and triangles are used to implement the interface `java.lang.Comparable`. The method `compareTo(o: Object): int` is used to compare objects of a class that implements this interface.
Writing a generic **max** Method

The **return** value from the **max** method is of the **Comparable** type. So, we need to cast it to **String** or **Date** explicitly.
Defining Classes to Implement Comparable

- We cannot use the `max` method to find the larger of two instances of `Rectangle`, because `Rectangle` does not implement `Comparable`.
- We can define a new rectangle class `ComparableRectangle` that implements `Comparable`: the instances of this new class are comparable.

```
GeometricObject
  
Rectangle
  
ComparableRectangle
  «interface»
  java.lang.Comparable
    +compareTo(o: Object): int
```
public class ComparableRectangle extends Rectangle
    implements Comparable {
    /** Construct a ComparableRectangle with specified properties */
    public ComparableRectangle(double width, double height) {
        super(width, height);
    }

    /** Implement the compareTo method defined in Comparable */
    public int compareTo(Object o) {
        if (getArea() > ((ComparableRectangle)o).getArea())
            return 1;
        else if (getArea() < ((ComparableRectangle)o).getArea())
            return -1;
        else
            return 0;
    }

    public static void main(String[] args) {
        ComparableRectangle rectangle1 = new ComparableRectangle(4, 5);
        ComparableRectangle rectangle2 = new ComparableRectangle(3, 6);
        System.out.println(Max.max(rectangle1, rectangle2));
    }
}
The static `sort` and `binarySearch` methods in the `java.util.Arrays` class uses the `Comparable` interface for arrays of Objects. For example:

```java
java.util.Arrays.sort(intArray);
```
The Cloneable Interface

- **Marker Interface**: is an empty interface (does not contain constants or methods), but it is used to denote that a class possesses certain desirable properties to the compiler and the JVM.

```java
package java.lang;
public interface Cloneable {
}
```

- A class that **implements** the **Cloneable** interface is marked cloneable:
  - its objects can be cloned using the `clone()` method defined in the `Object` class, and we can override this method in our classes
The Cloneable Interface

- **Calendar** (in the Java library) implements **Cloneable**:

  ```java
  Calendar calendar = new GregorianCalendar(2022, 1, 1);
  Calendar calendarCopy = (Calendar)(calendar.clone());
  System.out.println("calendar == calendarCopy is "+(calendar == calendarCopy));
  Displays:
  calendar == calendarCopy is false
  because the references are different
  System.out.println("calendar.equals(calendarCopy) is "+
calendar.equals(calendarCopy));
  calendar.equals(calendarCopy) is true
  because the calendarCopy is a copy of calendar
  ```
Implementing the **Cloneable** Interface

- If we try to create a clone of an object instance of a class that does not implement the **Cloneable** interface, it throws **CloneNotSupportedException**
- The **clone()** method in the **Object** class creates a new instance of the class of this object and initializes all its fields with exactly the contents of the corresponding fields of this object, as if by assignment (using a technique named **reflection**); the contents of the reference data fields are not cloned.
  - The **clone()** method returns an **Object** that needs to be casted
  - We can override the **clone()** method from the **Object** class to create custom clones
public class House implements Cloneable {
    private int id;
    private double area;
    private java.util.Date whenBuilt;
    public House(int id, double area) {
        this.id = id;
        this.area = area;
        whenBuilt = new java.util.Date();
    }
    public House(int id, double area, java.util.Date whenBuilt) {
        this.id = id; this.area = area; this.whenBuilt = whenBuilt;
    }
    public double getId() { return id; }
    public double getArea() { return area; }
    public java.util.Date getWhenBuilt() { return whenBuilt; }
    /** Override the protected clone method defined in the Object
     * class, and strengthen its accessibility */
    public Object clone() {
        // return new House(id,area,whenBuilt); // OR
        try {
            return super.clone();
        } catch (CloneNotSupportedException ex) {
            return null;
        }
    }
}
Shallow vs. Deep Copy

House house1 = new House(1, 1750.50);
House house2 = (House) (house1.clone());

**Shallow Copy:** If the field is of reference type, the object’s reference is copied rather than its content.
For *deep copying*, we can override the clone method with custom object creation:

```java
public class House implements Cloneable {
    ...
    public Object clone() { // deep copy
        try {
            House h = (House)(super.clone());
            h.whenBuilt = (Date)(whenBuilt.clone());
            return h;
        }catch (CloneNotSupportedException ex) {
            return null;
        }
    }
    ...
}
```
Interfaces vs. Abstract Classes

- In an interface, the data fields must be constants; an abstract class can have variable data fields
- Interfaces don't have constructors; all abstract classes have constructors
- Each method in an interface has only a signature without implementation (i.e., only abstract methods); an abstract class can have concrete methods

<table>
<thead>
<tr>
<th></th>
<th>Variables</th>
<th>Constructors</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interfaces</td>
<td>All variables must be public static final</td>
<td>No constructors.</td>
<td>All methods must be public abstract methods</td>
</tr>
<tr>
<td></td>
<td></td>
<td>An interface cannot be instantiated using the new operator.</td>
<td></td>
</tr>
<tr>
<td>Abstract classes</td>
<td>No restrictions</td>
<td>Constructors are invoked by subclasses through constructor chaining.</td>
<td>No restrictions.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>An abstract class cannot be instantiated using the new operator.</td>
<td></td>
</tr>
</tbody>
</table>
Interfaces Inheritance

- An interface can extend any number of other interfaces
- There is no root for interfaces
- A class can implement any number of interfaces
Conflicting interfaces

- Errors detected by the compiler:
  - If a class implements two interfaces with conflicting information, like:
    - two same constants with different values, or
    - two methods with same signature but different return type
Whether to use a class or an interface?

- **Strong is-a**: a relationship that clearly describes a parent-child relationship
  - For example: a student is a person
  - Should be modeled using class inheritance
- **Weak is-a (or is-kind-of)**: indicates that an object possesses a certain property
  - For example: all strings are comparable, all dates are comparable
  - Should be modeled using interfaces
- You can also use interfaces to circumvent single inheritance restriction if multiple inheritance is desired