Chapter 12 Abstract Classes and Interfaces

CS1: Java Programming
Colorado State University

Original slides by Daniel Liang
Modified slides by Kris Brown
Motivations

- You have learned how to write simple programs to create and display GUI components. Can you write the code to respond to user actions, such as clicking a button to perform an action?
- In order to write such code, you have to know about interfaces. An interface is for defining common behavior for classes (including unrelated classes). Before discussing interfaces, we introduce a closely related subject: abstract classes.
Abstract Classes and Abstract Methods

```
GeometricObject
- color: String
- filled: boolean
- dateCreated: java.util.Date

# GeometricObject()
# GeometricObject(color: string, filled: boolean)
+ getColor(): String
+ setColor(color: String): void
+ isFilled(): boolean
+ setFilled(filled: boolean): void
+ getDateCreated(): java.util.Date
+ toString(): String
+ getArea(): double
+ getPerimeter(): double
```

Methods `getArea` and `getPerimeter` are overridden in `Circle` and `Rectangle`. Superclass methods are generally omitted in the UML diagram for subclasses.

```
Circle
- radius: double

+ Circle()
+ Circle(radius: double)
+ Circle(radius: double, color: string, filled: boolean)
+ getRadius(): double
+ setRadius(radius: double): void
+ getDiameter(): double

Rectangle
- width: double
- height: double

+ Rectangle()
+ Rectangle(width: double, height: double)
+ Rectangle(width: double, height: double, color: string, filled: boolean)
+ getWidth(): double
+ setWidth(width: double): void
+ getHeight(): double
+ setHeight(height: double): void
```
abstract method in abstract class

An abstract method cannot be contained in a nonabstract class. If a subclass of an abstract superclass does not implement all the abstract methods, the subclass must be defined abstract. In other words, in a nonabstract subclass extended from an abstract class, all the abstract methods must be implemented, even if they are not used in the subclass.
An abstract class cannot be instantiated using the new operator, but you can still define its constructors, which are invoked in the constructors of its subclasses. For instance, the constructors of GeometricObject are invoked in the Circle class and the Rectangle class.
abstract class without abstract method

A class that contains abstract methods must be abstract. However, it is possible to define an abstract class that contains no abstract methods. In this case, you cannot create instances of the class using the new operator. This class is used as a base class for defining a new subclass.
superclass of abstract class may be concrete

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.
A subclass can override a method from its superclass to define it abstract. This is rare, but useful when the implementation of the method in the superclass becomes invalid in the subclass. In this case, the subclass must be defined abstract.
abstract class as type

You cannot create an instance from an abstract class using the new operator, but an abstract class can be used as a data type. Therefore, the following statement, which creates an array whose elements are of GeometricObject type, is correct.

GeometricObject[] geo = new GeometricObject[10];
Case Study: the Abstract Number Class

```java
java.lang.Number
+byteValue(): byte
+shortValue(): short
+intValue(): int
+longValue(): long
+floatValue(): float
+doubleValue(): double
```

```
Double | Float | Long | Integer | Short | Byte | BigInteger | BigDecimal
```

LargestNumbers  Run
Interfaces

What is an interface?
Why is an interface useful?
How do you define an interface?
How do you use an interface?
What is an interface?
Why is an interface useful?

An interface is a classlike construct that contains only constants and abstract methods. In many ways, an interface is similar to an abstract class, but the intent of an interface is to specify common behavior for objects. For example, you can specify that the objects are comparable, edible, cloneable using appropriate interfaces.
Define an Interface

To distinguish an interface from a class, Java uses the following syntax to define an interface:

```java
public interface InterfaceName {
    constant declarations;
    abstract method signatures;
}
```

Example:

```java
public interface Edible {
    /** Describe how to eat */
    public abstract String howToEat();
}
```
Interface is a Special Class

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, but in most cases 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 variable, as the result of casting, and so on.
Consider the task of writing classes to represent 2D shapes such as Circle, Rectangle, and Triangle.

There are certain attributes or operations that are common to all shapes: perimeter, area.

By being a Shape, you promise that you can compute those attributes, but each shape computes them differently.
Interface as a contract

- Analogous to the idea of roles or certifications in real life:
  - "I'm certified as a CPA accountant. The certification assures you that I know how to do taxes, perform audits."
  
  Compare to:
  - "I'm certified as a Shape. That means you can be sure that I know how to compute my area and perimeter."
The area and perimeter of shapes

- **Rectangle** (as defined by width $w$ and height $h$):
  
  \[ \text{area} = w \times h \]
  \[ \text{perimeter} = 2w + 2h \]

- **Circle** (as defined by radius $r$):
  
  \[ \text{area} = \pi r^2 \]
  \[ \text{perimeter} = 2\pi r \]

- **Triangle** (as defined by side lengths $a$, $b$, and $c$):
  
  \[ \text{area} = \sqrt{s(s-a)(s-b)(s-c)} \]
  \[ \text{where } s = \frac{1}{2}(a+b+c) \]
  \[ \text{perimeter} = a + b + c \]
Interfaces

- **interface**: A list of methods that a class promises to implement.
  - Inheritance encodes an is-a relationship and provides code-sharing.
    - An Executive object can be treated as a StaffMember, and Executive inherits StaffMember’s code.
  - An interface specifies what an object is capable of; no code sharing.
    - Only method **stubs** in the interface
    - Object **can-act-as** any interface it **implements**
    - A Rectangle does what you expect from a Shape as long as it implements the interface.
Java Interfaces

- An interface for shapes:
  ```java
  public interface Shape {
    public double area();
    public double perimeter();
  }
  ```
  - This interface describes the functionality common to all shapes. (Every shape knows how to compute its area and perimeter.)

- Interface declaration syntax:
  ```java
  public interface <name> {
    public <type> <name>(<type> <name>, . . . , <type> <name>);
    public <type> <name>(<type> <name>, . . . , <type> <name>);
    ...
    public <type> <name>(<type> <name>, . . . , <type> <name>);
  }
  ```
  - All methods are public!
public class Circle implements Shape {
    private double radius;

    // Constructs a new circle with the given radius.
    public Circle(double radius) {
        this.radius = radius;
    }

    // Returns the area of the circle.
    public double area() {
        return Math.PI * radius * radius;
    }

    // Returns the perimeter of the circle.
    public double perimeter() {
        return 2.0 * Math.PI * radius;
    }
}
Implementing an interface

- A class can declare that it implements an interface.
  - This means the class needs to contain an implementation for each of the methods in that interface.
    (Otherwise, the class will fail to compile.)

- Syntax for implementing an interface
  ```java
  public class <name> implements <interface name> {
      ...
  }
  ```
Requirements

- If we write a class that claims act like a `Shape` but doesn't implement the `area` and `perimeter` methods, it will not compile.

  - Example:
    ```java
    public class Banana implements Shape {
        // without implementing area or perimeter
    }
    ```

  - The compiler error message:
    ```java
    Banana.java:1: Banana is not abstract and does not override abstract method area() in Shape
    public class Banana implements Shape {
    ^
    ```
Diagramming an interface

- We draw arrows from the classes to the interface(s) they implement.
- Like inheritance, an interface represents an is-a relationship (a Circle is a Shape).
public class Rectangle implements Shape {
    private double width;
    private double height;

    // Constructs a new rectangle with the given dimensions.
    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    // Returns the area of this rectangle.
    public double area() {
        return width * height;
    }

    // Returns the perimeter of this rectangle.
    public double perimeter() {
        return 2.0 * (width + height);
    }
}
public class Triangle implements Shape {
    private double a;
    private double b;
    private double c;

    // Constructs a new Triangle given side lengths.
    public Triangle(double a, double b, double c) {
        this.a = a;
        this.b = b;
        this.c = c;
    }

    // Returns a triangle's area using Heron's formula.
    public double area() {
        double s = (a + b + c) / 2.0;
        return Math.sqrt(s * (s - a)*(s - b)*(s - c));
    }

    // Returns the perimeter of the triangle.
    public double double perimeter() {
        return a + b + c;
    }
}
Polymorphism is possible with interfaces.

Example:
```java
public static void printInfo(Shape s) {
    System.out.println("The shape: " + s);
    System.out.println("area : " + s.area());
    System.out.println("perim: " + s.perimeter());
    System.out.println();
}
```

Any object that implements the interface may be passed as the parameter to the above method.
```java
Circle circ = new Circle(12.0);
Triangle tri = new Triangle(5, 12, 13);
printInfo(circ);
printInfo(tri);
```
Interfaces and polymorphism

- We can create an array of an interface type, and store any object implementing that interface as an element.

```java
Circle circ = new Circle(12.0);
Rectangle rect = new Rectangle(4, 7);
Triangle tri = new Triangle(5, 12, 13);
Shape[] shapes = {circ, tri, rect};
for (int i = 0; i < shapes.length; i++) {
    printInfo(shapes[i]);
}
```

- Each element of the array executes the appropriate behavior for its object when it is passed to the `printInfo` method, or when `area` or `perimeter` is called on it.
Comments about Interfaces

- The term interface also refers to the set of public methods through which we can interact with objects of a class.
- Methods of an interface are abstract.
- Think of an interface as an abstract base class with all methods abstract
- Interfaces are used to define a contract for how you interact with an object, independent of the underlying implementation.
- Separate behavior (interface) from the implementation
Commonly used Java interfaces

- The Java class library contains several interfaces:
  - Comparable — allows us to order the elements of an arbitrary class
  - Serializable (in java.io) — for saving objects to a file.
  - List, Set, Map, Iterator (in java.util) — describe data structures for storing collections of objects
The Java Comparable interface

- A class can implement the `Comparable` interface to define an ordering for its objects.
  ```java
  public interface Comparable<E> {
      public int compareTo(E other);
  }
  ```
  
  ```java
  public class Employee implements Comparable<Employee> { ... }
  ```

- A call of `a.compareTo(b)` should return:
  - a value `< 0` if `a` comes "before" `b` in the ordering,
  - a value `> 0` if `a` comes "after" `b` in the ordering,
  - or `0` if `a` and `b` are considered "equal" in the ordering.
Comparable and sorting

If you implement Comparable, you can sort arbitrary objects using the method `Arrays.sort`

```java
StaffMember[] staff = new StaffMember[3];
staff[0] = new Executive(...);
staff[1] = new Employee(...);
staff[2] = new Hourly(...);
staff[3] = new Volunteer(...);
Arrays.sort(staff);
```

Note that you will need to provide an implementation of `compareTo`
compareTo tricks

- Delegation trick - If your object's attributes are comparable (such as strings), you can use their compareTo:

```java
// sort by employee name
public int compareTo(StaffMember other) {
    return name.compareTo(other.getName());
}
```
Example
You can now 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 (See TestEdible).

Notation:
The interface name and the method names are italicized. The dashed lines and hollow triangles are used to point to the interface.
Example: The **Comparable** Interface

```java
// This interface is defined in
// java.lang package
package java.lang;

public interface Comparable<E> {
    public int compareTo(E o);
}
```
The `toString`, `equals`, and `hashCode` Methods

Each wrapper class overrides the `toString`, `equals`, and `hashCode` methods defined in the `Object` class. Since all the numeric wrapper classes and the `Character` class implement the `Comparable` interface, the `compareTo` method is implemented in these classes.
Integer and BigInteger Classes

```java
public class Integer extends Number implements Comparable<Integer> {
    // class body omitted

    @Override
    public int compareTo(Integer o) {
        // Implementation omitted
    }
}
```

```java
public class BigInteger extends Number implements Comparable<BigInteger> {
    // class body omitted

    @Override
    public int compareTo(BigInteger o) {
        // Implementation omitted
    }
}
```

String and Date Classes

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

    @Override
    public int compareTo(String o) {
        // Implementation omitted
    }
}
```

```java
public class Date extends Object implements Comparable<Date> {
    // class body omitted

    @Override
    public int compareTo(Date o) {
        // Implementation omitted
    }
}
```
Example

1  System.out.println(new Integer(3).compareTo(new Integer(5)));
2  System.out.println("ABC".compareTo("ABE"));
3  java.util.Date date1 = new java.util.Date(2013, 1, 1);
4  java.util.Date date2 = new java.util.Date(2012, 1, 1);
5  System.out.println(date1.compareTo(date2));
Generic sort Method

Let $n$ be an Integer object, $s$ be a String object, and $d$ be a Date object. All the following expressions are true.

$n \text{ instanceof } \text{Integer} \\
\text{n instanceof Object} \\
\text{n instanceof Comparable} \\

s \text{ instanceof String} \\
\text{s instanceof Object} \\
\text{s instanceof Comparable} \\

d \text{ instanceof java.util.Date} \\
\text{d instanceof Object} \\
\text{d instanceof Comparable}

The java.util.Arrays.sort(array) method requires that the elements in an array are instances of Comparable $<E>$. 

SortComparableObjects Run
Defining Classes to Implement Comparable

```java
GeometricObject

Rectangle

ComparableRectangle

```
Interfaces vs. Abstract Classes

In an interface, the data must be constants; an abstract class can have all types of data.

Each method in an interface has only a signature without implementation; an abstract class can have concrete methods.

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

All classes share a single root, the Object class, but there is no single root for interfaces. Like a class, an interface also defines a type. A variable of an interface type can reference any instance of the class that implements the interface. If a class extends an interface, this interface plays the same role as a superclass. You can use an interface as a data type and cast a variable of an interface type to its subclass, and vice versa.

Suppose that c is an instance of Class2. c is also an instance of Object, Class1, Interface1, Interface1_1, Interface1_2, Interface2_1, and Interface2_2.
Whether to use an interface or a class?

Abstract classes and interfaces can both be used to model common features. How do you decide whether to use an interface or a class? In general, a strong is-a relationship that clearly describes a parent-child relationship should be modeled using classes. For example, a staff member is a person. A weak is-a relationship, also known as an is-kind-of relationship, indicates that an object possesses a certain property. A weak is-a relationship can be modeled using interfaces. For example, all strings are comparable, so the String class implements the Comparable interface. You can also use interfaces to circumvent single inheritance restriction if multiple inheritance is desired. In the case of multiple inheritance, you have to design one as a superclass, and others as interface.
The **Rational Class**

```java
java.lang.Number
Rational
java.lang.Comparable<Rational>
```

**Rational**

- numerator: long
- denominator: long

```java
+Rational()
+Rational(numerator: long, denominator: long)
+getNumerator(): long
+getDenominator(): long
+add(secondRational: Rational): Rational
+subtract(secondRational: Rational): Rational
+multiply(secondRational: Rational): Rational
+divide(secondRational: Rational): Rational
+toString(): String
-gcd(n: long, d: long): long
```

**The numerator of this rational number.**
**The denominator of this rational number.**

Creates a rational number with numerator 0 and denominator 1.
Creates a rational number with a specified numerator and denominator.

Returns the numerator of this rational number.
Returns the denominator of this rational number.

Returns the addition of this rational number with another.
Returns the subtraction of this rational number with another.

Returns the multiplication of this rational number with another.
Returns the division of this rational number with another.

Returns a string in the form “numerator/denominator.” Returns the numerator if denominator is 1.
Returns the greatest common divisor of n and d.
Designing a Class, cont.

Provide a public no-arg constructor and override the `equals` method and the `toString` method defined in the `Object` class whenever possible.
Designing a Class, cont.

Follow standard Java programming style and naming conventions. Choose informative names for classes, data fields, and methods. Always place the data declaration before the constructor, and place constructors before methods. Always provide a constructor and initialize variables to avoid programming errors.
Misc Slides
Designing a Class

(Coherence) A class should describe a single entity, and all the class operations should logically fit together to support a coherent purpose. You can use a class for students, for example, but you should not combine students and staff in the same class, because students and staff have different entities.
Designing a Class, cont.

(Separating responsibilities) A single entity with too many responsibilities can be broken into several classes to separate responsibilities. The classes String, StringBuilder, and StringBuffer all deal with strings, for example, but have different responsibilities. The String class deals with immutable strings, the StringBuilder class is for creating mutable strings, and the StringBuffer class is similar to StringBuilder except that StringBuffer contains synchronized methods for updating strings.
Designing a Class, cont.

Classes are designed for reuse. Users can incorporate classes in many different combinations, orders, and environments. Therefore, you should design a class that imposes no restrictions on what or when the user can do with it, design the properties to ensure that the user can set properties in any order, with any combination of values, and design methods to function independently of their order of occurrence.
Using Visibility Modifiers

Each class can present two contracts – one for the users of the class and one for the extenders of the class. Make the fields private and accessor methods public if they are intended for the users of the class. Make the fields or method protected if they are intended for extenders of the class. The contract for the extenders encompasses the contract for the users. The extended class may increase the visibility of an instance method from protected to public, or change its implementation, but you should never change the implementation in a way that violates that contract.
Using Visibility Modifiers, cont.

A class should use the private modifier to hide its data from direct access by clients. You can use get methods and set methods to provide users with access to the private data, but only to private data you want the user to see or to modify. A class should also hide methods not intended for client use. The gcd method in the Rational class is private, for example, because it is only for internal use within the class.
Using the static Modifier

A property that is shared by all the instances of the class should be declared as a static property.
Objectives

- To design and use abstract classes (§13.2).
- To generalize numeric wrapper classes, \texttt{BigInteger}, and \texttt{BigDecimal} using the abstract \texttt{Number} class (§13.3).
- To process a calendar using the \texttt{Calendar} and \texttt{GregorianCalendar} classes (§13.4).
- To specify common behavior for objects using interfaces (§13.5).
- To define interfaces and define classes that implement interfaces (§13.5).
- To define a natural order using the \texttt{Comparable} interface (§13.6).
- To make objects cloneable using the \texttt{Cloneable} interface (§13.7).
- To explore the similarities and differences among concrete classes, abstract classes, and interfaces (§13.8).
- To design the \texttt{Rational} class for processing rational numbers (§13.9).
- To design classes that follow the class-design guidelines (§13.10).
The Abstract Calendar Class and Its GregorianCalendar subclass

```java
java.util.Calendar

- Calendar()
- get(field: int): int
- set(field: int, value: int): void
- set(year: int, month: int, dayOfMonth: int): void
- getActualMaximum(field: int): int
- add(field: int, amount: int): void
- getTime(): java.util.Date
- setTime(date: java.util.Date): void

Constructs a default calendar.
Returns the value of the given calendar field.
Sets the given calendar to the specified value.
Sets the calendar with the specified year, month, and date. The month parameter is 0-based; that is, 0 is for January.
Returns the maximum value that the specified calendar field could have.
Adds or subtracts the specified amount of time to the given calendar field.
Returns a Date object representing this calendar’s time value (million second offset from the UNIX epoch).
Sets this calendar’s time with the given Date object.

java.util.GregorianCalendar

- GregorianCalendar()
- GregorianCalendar(year: int, month: int, dayOfMonth: int)
- GregorianCalendar(year: int, month: int, dayOfMonth: int, hour: int, minute: int, second: int)

Constructs a GregorianCalendar for the current time.
Constructs a GregorianCalendar for the specified year, month, and date.
Constructs a GregorianCalendar for the specified year, month, date, hour, minute, and second. The month parameter is 0-based, that is, 0 is for January.
```
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. Subclasses of Calendar can implement specific calendar systems such as Gregorian calendar, Lunar Calendar and Jewish calendar. Currently, java.util.GregorianCalendar for the Gregorian calendar is supported in the Java API.
The GregorianCalendar Class

You can use new GregorianCalendar() to construct a default GregorianCalendar with the current time and use new GregorianCalendar(year, month, date) to construct a GregorianCalendar with the specified year, month, and date. The month parameter is 0-based, i.e., 0 is for January.
The get Method in 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, 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, with 1 for the first week.</td>
</tr>
<tr>
<td>WEEK_OF_YEAR</td>
<td>The week number within the year, with 1 for the first week.</td>
</tr>
<tr>
<td>AM_PM</td>
<td>Indicator for AM or PM (0 for AM and 1 for PM).</td>
</tr>
</tbody>
</table>
Getting Date/Time Information from Calendar