Chapter 13 Abstract Classes and Interfaces

Motivations

▪ We have learned how to write simple programs to create and display GUI components. Can we write the code to respond to user actions, such as clicking a button to perform an action?

▪ In order to write such code, we 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.
Objectives

- To design and use abstract classes (§13.2).
- To generalize numeric wrapper classes, `BigInteger`, and `BigDecimal` using the abstract `Number` class (§13.3).
- To process a calendar using the `Calendar` and `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 `Comparable` interface (§13.6).
- To make objects cloneable using the `Cloneable` interface (§13.7).
- To explore the similarities and differences among concrete classes, abstract classes, and interfaces (§13.8).
- To design the `Rational` class for processing rational numbers (§13.9).
- To design classes that follow the class-design guidelines (§13.10).

Abstract Classes & Abstract Methods

- An abstract class is a class that is declared abstract
- Abstract classes cannot be instantiated, but they can be subclassed
- Abstract methods are declared when their implementations are dependent on the subclasses where they are invoked, e.g. `getArea()` in `GeometricObject`
- Abstract methods are non-static methods that are declared without implementations (without `{}`, but followed by a `;`):

```java
public abstract class Account {
    // declare data fields
    // declare concrete methods
    abstract void check(double balance);
}
```
Abstract Classes & Abstract Methods

- In a concrete subclass extended from an abstract class, all the abstract methods must be implemented, even if they are not used in the subclass.

- If a subclass of an abstract superclass does not implement all the abstract methods, the subclass must be defined abstract.

- An abstract class may contain just concrete methods, or just abstract methods, or both or none!

- A concrete class cannot contain abstract methods.
Abstract Classes

- 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.

- We cannot create an instance from an abstract class using the `new` operator, but:
  - We can include constructors (to be used by subclasses) in an abstract class.
  - An abstract class can be used as a data type. For example, the following statement, which creates an array whose elements are of abstract class `GeometricObject` type, is correct:

```java
GeometricObject[] geo = new GeometricObject[10];
```

Case Study: Abstract Number Class

```
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
Abstract Calendar Class & its Concrete 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 Concrete Class

- We can use `new GregorianCalendar()` to construct a default `GregorianCalendar` object with the current time.

- We can also use `new GregorianCalendar(year, month, date)` to construct a `GregorianCalendar` object with the specified year, month, and date.

- The month parameter is 0-based, i.e., 0 is for January.

get() in Calendar Class

`get(int field)` is used to extract detailed information from `Calendar` objects. The possible fields are:

<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>
Interfaces

- The Apple & Chicken classes are based on different superclasses, but both share a common behavior: both are edible! We can model this common behavior using an interface

- Interfaces can be used to specify common behavior for objects of unrelated classes as well as related classes

- Abstract classes model the common behavior of related classes only

interface & class Similarities

- 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, we cannot create an instance from an interface using the new operator, but in most cases we can use an interface more or less the same way we use an abstract class
  - For example, we can use an interface as a data type and cast a variable of an interface type to its subclass, and vice versa
Defining an **interface**

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

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

**Example:**
```java
public interface Edible {
    /** Describe how to eat */
    public abstract String howToEat();
}
```

**interface inheritance**

We 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 (or **inherit** the interface) using the `implements` keyword

**Example:** Chicken & Fruit implement the Edible interface in TestEdible
Omitting Modifiers in Interfaces

All data fields are public final static and all methods are public abstract in an interface. For this reason, these modifiers can be omitted, as shown below:

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

Equivalent

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

A constant defined in an interface can be accessed using syntax InterfaceName.CONSTANT_NAME (e.g., T1.K)

Example: The Comparable Interface

This interface defines compareTo() for comparing objects

```java
package java.lang;

public interface Comparable<E> {
    public int compareTo(E o);
}
```

compareTo() determines the order of this object with the specified object o and returns a negative integer, zero, or a positive integer if this object is less than, equal to, or greater than o
**compareTo() in `Integer`, `BigInteger`, `String` & `Date`**

All numeric wrapper classes, `Character` and `String` implement `Comparable`, thus `compareTo()` is implemented in all of them.

```java
public class Integer extends Number implements Comparable<Integer> {
  // class body omitted
  @Override
  public int compareTo(Integer o) {
    // Implementation omitted
  }
}

public class BigInteger extends Number implements Comparable<BigInteger> {
  // class body omitted
  @Override
  public int compareTo(BigInteger o) {
    // Implementation omitted
  }
}

public class String extends Object implements Comparable<String> {
  // class body omitted
  @Override
  public int compareTo(String o) {
    // Implementation omitted
  }
}

public class Date extends Object implements Comparable<Date> {
  // class body omitted
  @Override
  public int compareTo(Date o) {
    // Implementation omitted
  }
}
```

**Example: `compareTo()`**

```java
System.out.println(new Integer(3).compareTo(new Integer(5)));  // -2
System.out.println("ABC".compareTo("ABE"));  // -1
java.util.Date date1 = new java.util.Date(2013, 1, 1);
java.util.Date date2 = new java.util.Date(2012, 1, 1);
System.out.println(date1.compareTo(date2));  // -1
```

Displays:

```
-1
-2
1
```
The Comparable Interface

Let \( n \) be an \texttt{Integer} object, \( s \) be a \texttt{String} object and \( d \) be a \texttt{Date} object. Then, all the following expressions are true:

\[
\begin{align*}
\text{n instanceof Integer} \\
\text{n instanceof Object} \\
\text{n instanceof Comparable} \\
\text{s instanceof String} \\
\text{s instanceof Object} \\
\text{s instanceof Comparable} \\
\text{d instanceof java.util.Date} \\
\text{d instanceof Object} \\
\text{d instanceof Comparable}
\end{align*}
\]

Generic sort()

\texttt{java.util.Arrays.sort(array)} requires that the elements in an array implement the \texttt{Comparable<E>} interface.
Defining Classes to Implement Comparable

- We cannot use `sort()` to sort an array of `Rectangle` objects, because `Rectangle` does not implement `Comparable`.
- However, we can define a new rectangle class that implements `Comparable`. The instances of this new class are comparable.

```java
package java.lang;
public interface Comparable {
    int compareTo(Object o);
}
```

The Cloneable Interface

- The `Cloneable` interface specifies that an object can be cloned.
- A class that implements the `Cloneable` interface is marked `cloneable`, and its objects can be cloned using `clone()` defined in the `Object` class. `clone()` returns the copy of an object.

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

`Cloneable` is a marker interface, an empty interface:
- A marker interface does not contain constants or methods.
- It is used to denote that a class possesses certain desirable properties.
Example: Cloneable

Several classes (e.g., Date, Calendar and ArrayList) in the Java library implement Cloneable. Thus, the instances of these classes can be cloned. For example, the following code:

```java
Calendar calendar = new GregorianCalendar(2003, 2, 1);
Calendar calendarCopy = (Calendar) calendar.clone();
System.out.println("calendar == calendarCopy is " +
(calendar == calendarCopy));
System.out.println("calendar.equals(calendarCopy) is " +
calendar.equals(calendarCopy));
```

displays

```
calendar == calendarCopy is false
calendar.equals(calendarCopy) is true
```

Implementing Cloneable Interface

To define a custom class that implements the Cloneable interface, the class must override `clone()` in `Object`

The following code defines a class named `House` that implements Cloneable and Comparable

```java
House
```
Shallow vs. Deep Copy

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

If object A is copied as object B then if any properties of A are a reference to a memory location, at the end of copying the two objects will be using the same reference. The default clone() makes a shallow copy.

Instead of references, if the actual values are copied to a new object then that will be a deep copy. This way the two objects are completely independent. To make a deep copy, we need to override the default clone()
Interfaces vs. Abstract Classes

<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 (abstract methods can’t be static)</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>Interface</td>
<td>Must be public static final</td>
<td>Must be public abstract</td>
</tr>
<tr>
<td></td>
<td>No constructors. An interface cannot be instantiated using the new operator</td>
<td>(abstract methods can’t be static)</td>
</tr>
</tbody>
</table>

Interfaces

- All classes share a single root (Object). There is no single root for interfaces
- A variable of an interface type can reference any instance of the class that implements that interface
- Sub-interfaces: public interface infB extends infA {...}

If c is an instance of Class2, c is also an instance of Object, Class1, Interface1, Interface1_1, Interface1_2, Interface2_1, and Interface2_2
## The Rational Class

The Rational class is defined in the `java.lang` package and implements the `java.lang.Comparable` and `java.lang.Runnable` interfaces. It provides methods for arithmetic operations on rational numbers and supports the `Comparable` interface for natural ordering of rational numbers.

### Class Design Guidelines: Coherence

- A class should describe a single entity, and all the class operations should logically fit together to support a coherent purpose.
  - We can use a class for students, for example, but we should not combine students and staff in the same class, because students and staff are different entities.

- A single entity with many responsibilities should be broken into several classes to separate the responsibilities.
  - The classes `String`, `StringBuilder`, and `StringBuffer` all deal with strings, for example, but have different responsibilities:
    - `String` deals with immutable strings
    - `StringBuilder` deals with mutable strings
    - `StringBuffer` is similar to `StringBuilder` except that it contains synchronized methods that are required for threaded applications.
Class Design Guidelines: **Consistency**

- Follow standard Java *programming style* and *naming conventions*
  - Choose informative names for classes, data fields, and methods
  - Place the data declaration before the constructor and place constructors before methods
  - Make the names consistent. For example, `length()` returns the size of a `String`, `StringBuilder`, or `StringBuffer`. It would not be consistent if different names were used for this method in these classes

- Provide a **public no-arg constructor** for constructing an instance
  - If a class does not support a *no-arg constructor*, document the reason
  - If no constructors are defined explicitly, a **public no-arg constructor** with an empty body is assumed

- Override `equals()`, `hashCode()` and `toString()` defined in the `Object` class whenever possible

Class Design Guidelines: **Encapsulation**

- A class should use the **private** modifier to hide its data from direct access by clients. This makes the class easy to maintain

- Provide **accessors** and **mutators** only if necessary
  - **Example:** The `Rational` class provides accessors for `numerator` and `denominator`, but no mutators, because a `Rational` object is immutable
Class Design Guidelines: **Clarity**

**Cohesion, consistency, & encapsulation** improve class clarity. A class should also have a clear **contract** that is easy to explain & understand

- Users should be able to use classes in many different combinations, orders, and environments. Therefore, design a class that
  - imposes no restrictions on **how or when** the user can use it
  - allows the user to set properties in any order and with any combination of values
  - executes methods correctly independent of their order of occurrence

- Methods should be defined **intuitively** without causing confusion
  - **Example:** It is more intuitive to return a substring from `beginIndex` to `endIndex`, instead of `beginIndex` to `endIndex-1`

- Do not declare a data field that can be **derived** from other data fields (e.g. no need to store both `birthdate` and `age`)

---

Class Design Guidelines: **Completeness**

Classes are designed for use by **many different customers**. In order to be useful in a wide range of applications, a class should provide a **variety of ways for customization** through properties and methods

**Example:** The `String` class contains more than 40 methods that are useful for a variety of applications
Class Design Guidelines: **Instance vs. static**

- A variable or method that is dependent on a specific instance of the class must be an instance variable or method.
- A variable that is shared by all the instances of a class should be declared `static`.
- A method that is not dependent on a specific instance should be defined as a `static` method.
- Always reference `static` variables and methods from a class name to improve readability and avoid errors.
- A `static` variable or method can be invoked from an instance method, but an instance variable or method cannot be invoked from a `static` method.

---

Class Design Guidelines: **Inheritance vs. Aggregation**

The difference between inheritance and aggregation is the difference between an *is-a* and a *has-a* relationship:

- An apple *is a* fruit; thus, we would use inheritance to model the relationship between the classes `Apple` and `Fruit`.
- A person *has a* name; thus, we would use aggregation to model the relationship between the classes `Person` and `Name`. 
Class Design Guidelines: **Interfaces vs. Abstract Classes**

- Both can be used to specify common behavior. In general, a strong *is-a* relationship that clearly describes a parent-child relationship should be modeled using classes, e.g. orange *is a fruit*

- A weak *is-a* (or *is-kind-of*) relationship, indicates that an object possesses a certain property. A weak *is-a* relationship can be modeled using interfaces, e.g. all strings are comparable, so the `String` class implements the `Comparable` interface

- A circle or a rectangle is a geometric object, so `Circle` can be designed as a subclass of `GeometricObject`. Circles are different and comparable based on their radii, so `Circle` can implement the `Comparable` interface

- A subclass can extend only one superclass but can implement any number of interfaces

Class Design Guidelines: **Visibility Modifiers**

- Each class can present two contracts – one for the users of the class and one for the extenders of the class

- The contract for the extenders includes the contract for the users

- Make the fields private and accessor methods public if they are intended for the users of the class

- Make the fields or methods protected if they are intended for extenders of the class

- The extended class may increase the visibility of an instance method from protected to public, or change its implementation, but we should never change the implementation in a way that violates the contract