Interfaces

Introduction to Programming and Computational Problem Solving - 2

CSE 8B

Lecture 16

Announcements

- Assignment 7 is due tomorrow, 11:59 PM
  - Upgrade beginning Jun 4, 12:01 AM
- Assignment 8 will be released today
  - Due Jun 7, 11:59 PM
- Educational research study
  - Jun 2, weekly survey
  - Jun 5, post-test

Abstract classes and interfaces

- Remember, a superclass defines common behavior for **related** subclasses
- An *interface* can be used to define common behavior for classes, including **unrelated** classes
- Interfaces and abstract classes (covered last lecture) are closely related to each other

Abstract classes

- Remember, inheritance enables you to define a general class (i.e., a *superclass*) and later extend it to more specialized classes (i.e., *subclasses*)
- Sometimes, a superclass is so general it cannot be used to create objects
  - Such a class is called an *abstract class*
- An **abstract** class can contain abstract methods that are implemented in **concrete** subclasses
- Just like nonabstract classes, models **is-a** relationships
  - For example
    - Circle **is-a** GeometricObject
    - Rectangle **is-a** GeometricObject

Abstract class as a data type

- Remember, an abstract class cannot be instantiated using the new operator
- However, an abstract class can be used as a data type
  - Example

```
GeometricObject[] objects = new GeometricObject[2];
objects[0] = new Circle();
objects[1] = new Rectangle();
```

Abstract classes and interfaces

- An abstract class can contain abstract methods that are implemented in concrete subclasses
- An interface is a class-like construct that contains 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 and/or cloneable using appropriate interfaces

Defining an interface

- To distinguish an interface from a class, Java uses the keyword `interface`
  - The syntax to define an interface is

    ```
    public interface InterfaceName {
      // Constant declarations
      // Abstract method signatures
    }
    ```

  - Example

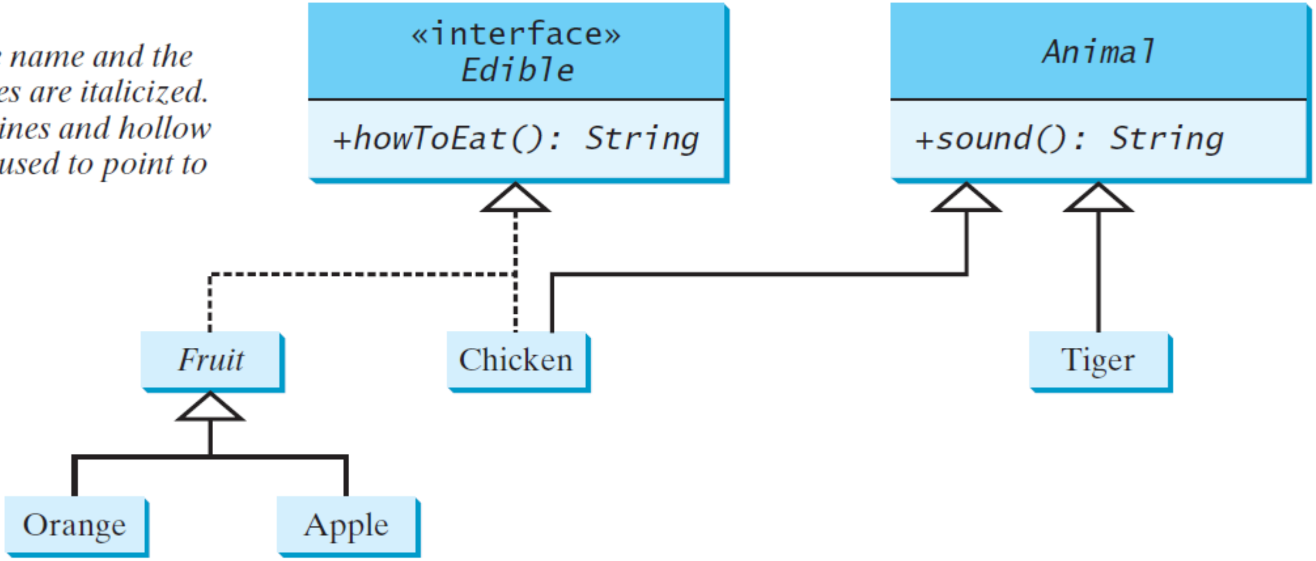
    ```
    public interface Edible {
      // Describe how to eat
      public abstract String howToEat();
    }
    ```

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
- Naming convention
  - Class names are nouns
  - Interface names may be adjectives or nouns
- Interfaces model **is-kind-of** relationships
  - For example
    - Fruit **is-kind-of** Edible

Interface example

- Use the `Edible` interface to specify whether an object is edible



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

Interfaces

- The class for the object implementing an interface uses the keyword `implements`
  - Examples

```
abstract class Fruit implements Edible {
  // Data fields, constructors, and methods
}

class Chicken extends Animal implements Edible {
  // Data fields, constructors, and methods
}
```

- The relationship between the class and the interface is known as *interface inheritance*

Omitting modifiers in interfaces

- **All data fields** are `public final static` and **all methods** are `public abstract` in an interface
  - As such, 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();
}
```

Interface static members

- Interfaces can have static members
- Like class static members, the best practice is to make invocations of static methods and access of static data fields obvious
- Use

```
InterfaceName.methodName(arguments)
InterfaceName.variable
```

Interface default methods

- A default method provides a default implementation for the method in the interface
  - Use the keyword `default`
  - Example

```
public interface A {
  public default void doSomething() {
    System.out.println("Do something");
  }
  ...
}
```

- A class that implements the interface may simply use the default implementation for the method or override the method with a new implementation

Interface example

- The `java.lang.Comparable` interface defines the `compareTo` method for comparing objects

```
package java.lang;

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

- The `compareTo` method returns
  - A negative integer if this object is less than o
  - Zero if this object is equal to o
  - A positive integer if this object is greater than o

The `Comparable` interface

- Many classes (e.g., the numeric wrapper classes) in the Java library implement `Comparable` to define a natural order for objects
  - The `compareTo` method is implemented in these classes

The `Comparable` interface

```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
  }
}
```

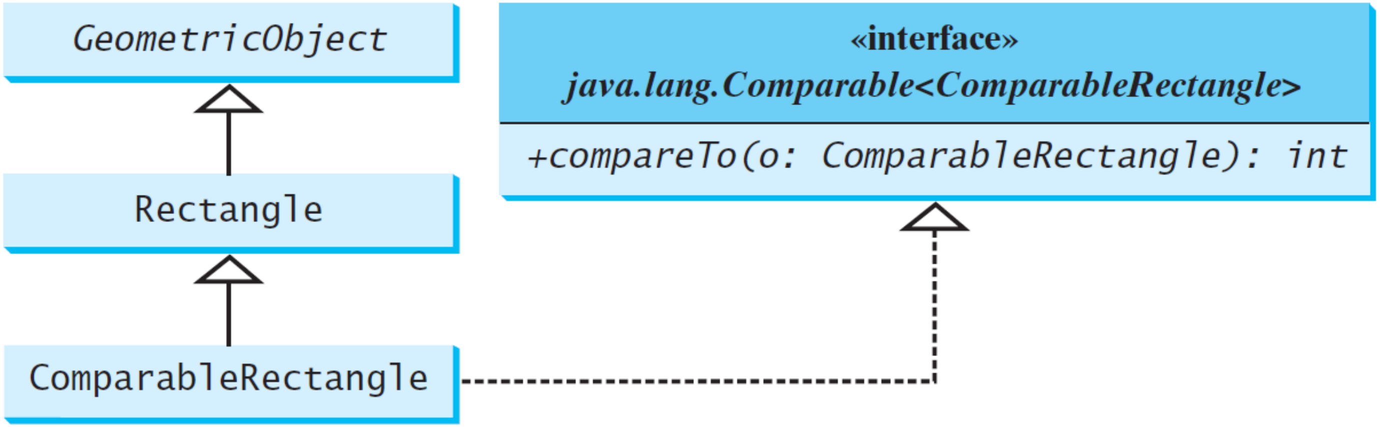
```java
public class String extends Object
    implements Comparable<String> {
  // class body omitted

  @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
  }
}
```

Defining classes to implement Comparable


GeometricObject
Rectangle
ComparableRectangle
«interface»
java.lang.Comparable<ComparableRectangle>
+compareTo(o: ComparableRectangle): int

```
public class ComparableRectangle extends Rectangle
    implements Comparable<ComparableRectangle> {
  // Construct a ComparableRectangle with specified properties
  public ComparableRectangle(double width, double height) {
    super(width, height);
  }

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

Interface example

- The `java.lang.Cloneable` interface specifies that an object can be cloned (i.e., it can be copied)

```
package java.lang;

public interface Cloneable {
}
```

- The interface is empty
  - An interface with an empty body is called a *marker interface*
- A class that implements the `Cloneable` interface is **marked** cloneable
  - Its objects can be cloned using the `clone` method **defined in the `Object` class**

The `Cloneable` interface

- Like `Comparable`, many classes in the Java library implement `Cloneable`
  - The instances of these classes can be cloned
  - Examples

Explicit casting

```
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
```

The `Cloneable` interface

- Arrays are cloneable
  - You can clone an array using the `clone` method

    ```
    int[] list1 = {1, 2};
    int[] list2 = list1.clone();
    ```

  - `ArrayList` implements `Cloneable`

Defining classes to implement `Cloneable`

- A class that implements the `Cloneable` interface must override the `clone` method defined in the `Object` class

```
protected native Object clone() throws CloneNotSupportedException;
```

- The keyword `native` indicates this method is not written in Java
  - It is implemented in the JVM for the native platform
- The class must override the `clone` method and change the visibility modifier to `public`, so it can be used in any package
- The class must implement `Cloneable`
  - Otherwise, `CloneNotSupportedException` is thrown

Defining classes to implement `Cloneable`

- To perform a *shallow copy*, the clone method in a class that implements the `Cloneable` interface can simply invoke the `super.clone` method

```
public abstract class GeometricObject
  implements Cloneable
{
  private String color;
  private boolean filled;
  private java.util.Date dateCreated;
  ...
  @Override
  public Object clone() throws CloneNotSupportedException {
    GeometricObject go = (GeometricObject)super.clone(); // Shallow copy
    return go;
  }
}
```

Shallow copy: for data fields that are objects, the objects' **references** are copied

Defining classes to implement `Cloneable`

- To perform a *deep copy*, the clone method in a class that implements the `Cloneable` interface must copy the contents of data fields that are objects

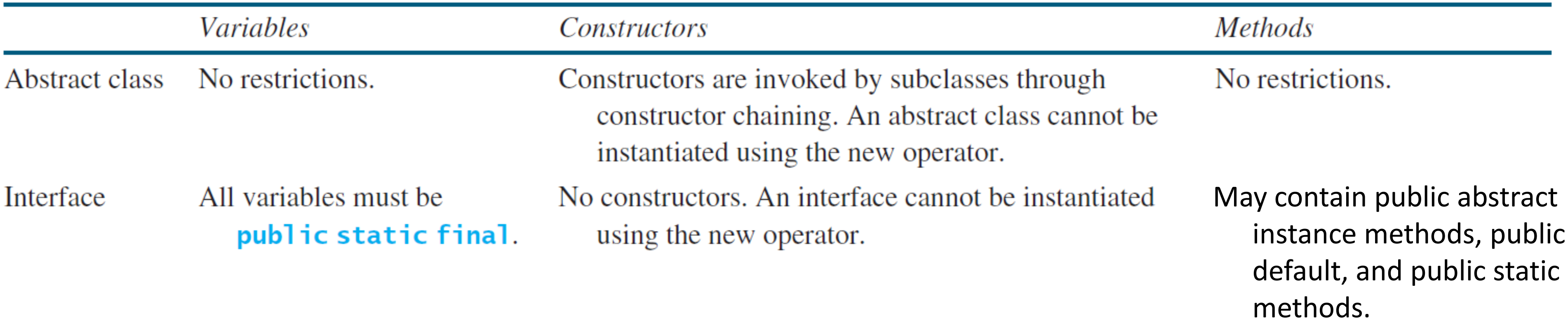
```
public abstract class GeometricObject
  implements Cloneable
{
  private String color;
  private boolean filled;
  private java.util.Date dateCreated;
  ...
  @Override
  public Object clone() throws CloneNotSupportedException {
    GeometricObject go = (GeometricObject)super.clone(); // Shallow copy
    // String is immutable, so deep copy is not required
    go.dateCreated = (java.util.Date)dateCreated.clone();
    return go;
  }
}
```

Shallow copy: for data fields that are objects, the objects' **references** are copied

Deep copy

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 (except default and static methods); an abstract class can have concrete methods



	Variables	*Constructors*	*Methods*
Abstract class	No restrictions.	Constructors are invoked by subclasses through constructor chaining. An abstract class cannot be instantiated using the new operator.	No restrictions.
Interface	All variables must be `public static final`.	No constructors. An interface cannot be instantiated using the new operator.	May contain public abstract instance methods, public default, and public static methods.

Interfaces vs. abstract classes

- An interface can inherit other interfaces using the `extends` keyword. Such an interface is called a *subinterface*.

```
public interface NewInterface
extends Interface1, ..., InterfaceN {
  // constants and abstract methods
}
```

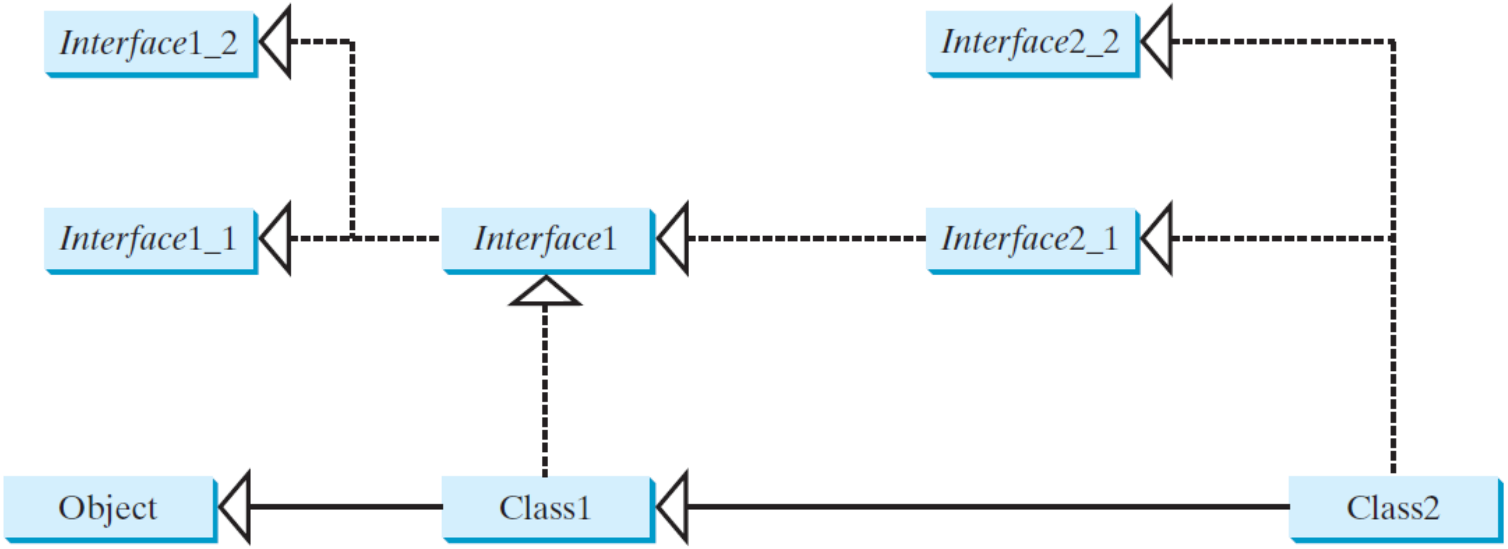
- A class implementing `NewInterface` must implement the abstract methods defined in `NewInterface`, `Interface1`, ..., and `InterfaceN`.
- An interface can extend other interfaces, but not classes

Interfaces vs. abstract classes

- 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 interface 2 extends interface 1, then interface 1 is like a superclass for interface 2
- You can use an interface as a data type and cast a variable of an interface type to its subclass, and vice versa

Interfaces vs. abstract classes

- **A class can implement multiple interfaces, but it can only extend one superclass**
- 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`



Conflicting interfaces

- On rare occasion, a class may implement two interfaces with conflicting information (e.g., two same constants with different values or two methods with same signature but different return type)
- This type of errors will be detected by the compiler

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
- A single entity with many responsibilities can be broken into several classes to separate responsibilities

Consistency

- Follow standard Java programming style and naming conventions
- Choose informative names for classes, data fields, and methods
  - Make names consistent
- Place the data declaration before the constructor, and place constructors before methods
- Provide a no-arg constructor (or document why the class does not support one)

Encapsulation

- A class should use the `private` modifier to hide its data from direct access by clients
- Provide getter methods and setter 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
- Make methods `protected` if they are intended for extenders of the class

Clarity and completeness

- A class should have a clear contract that is easy to explain and easy to understand
- Design a class that imposes no restrictions on how or when the user can use it
  - Design the properties in a way that lets the user set them in any order and with any combination of values
  - Design methods that function independently of their order of occurrence
- Methods should be defined intuitively without causing confusion
- You should not declare a data field that can be derived from other data fields
- A class should provide a variety of ways for customization through properties and methods that, together, are **minimal and complete**

Instance vs. static

- A variable or method dependent on a specific instance of the class must be an instance variable or method
- A variable shared by all the instances of a class should be declared static
- A method 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
- Do not initialize a static data field from a constructor parameter
  - Use a setter method to change the static data field

Inheritance vs. aggregation

- Use inheritance to model **is-a** relationships
- Use aggregation (and composition) to model **has-a** relationships

Interfaces vs. abstract classes

- Abstract classes and interfaces can both be used to model common behavior for objects
  - Interfaces cannot contain data fields, only constants
- In general, a strong **is-a** relationship clearly describes a parent-child relationship should be modeled using **classes**
- An **is-kind-of** relationship indicates an object possesses a certain property and can be modeled using **interfaces**
  - An interface can define a common supertype for **unrelated classes**
- **A subclass can extend only one superclass, but can implement any number of interfaces**
- You can also use interfaces to circumvent single inheritance restriction if multiple inheritance is desired
  - You must design one as a superclass, and others as interface

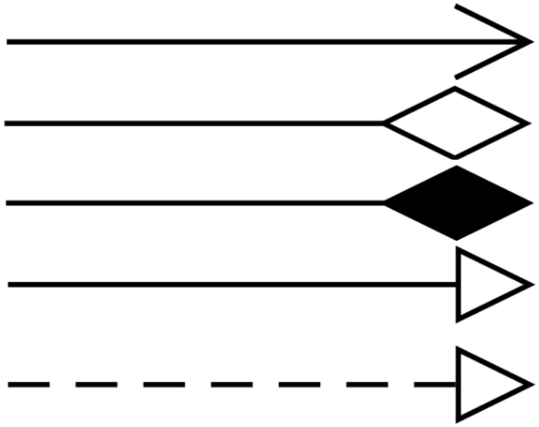
Unified Modeling Language (UML)

+ public

# protected

- private

- Static variables and methods are underlined
- Abstract class names and methods are *italicized*
- Interface names and methods are *italicized*
- Open or no arrow is association
- Hollow diamond is aggregation
- Filled diamond is composition
- Hollow triangle is inheritance
- Dashed line with hollow triangle is implementation of interface



Next Lecture

- Assertions and career day
  - Participation is optional
  - No prelecture quiz or lecture worksheet
  - Not on final exam
- Educational research study
  - Post-test