Lecture 13
Interfaces

What is an interface? Why is an interface useful?

- An interface is a class-like 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 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:

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

Example:

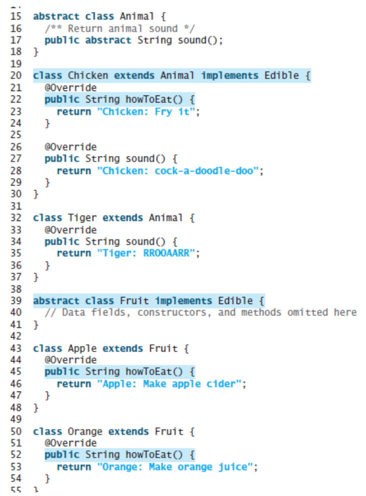
```
public interface Edible {
  /** Describe how to eat */
  public abstract String howToEat();
}
```

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.

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).



```java
abstract class Animal {
  /** Return animal sound */
  public abstract String sound();
}

class Chicken extends Animal implements Edible {
  @Override
  public String howToEat() {
    return "Chicken: Fry it";
  }

  @Override
  public String sound() {
    return "Chicken: cock-a-doodle-doo";
  }
}

class Tiger extends Animal {
  @Override
  public String sound() {
    return "Tiger: RROOAARR";
  }
}

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

class Apple extends Fruit {
  @Override
  public String howToEat() {
    return "Apple: Make apple cider";
  }
}

class Orange extends Fruit {
  @Override
  public String howToEat() {
    return "Orange: Make orange juice";
  }
}
```

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

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

public interface Comparable {
  public int compareTo(Object o);
}
```

String and Date Classes

- Many classes (e.g., String and Date) in the Java library implement Comparable to define a natural order for the objects.
- If you examine the source code of these classes, you will see the keyword 'implements' used in the classes, as shown below:

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

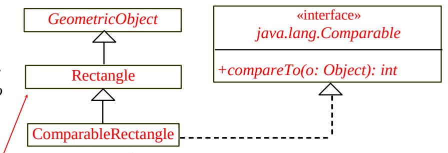
```
public class Date extends Object
    implements Comparable {
  // class body omitted
}
```

```
new String() instanceof String
new String() instanceof Comparable
new java.util.Date() instanceof java.util.Date
new java.util.Date() instanceof Comparable
```

9

Defining Classes to Implement Comparable

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



You cannot use the max method to find the larger of two instances of Rectangle, because Rectangle does not implement Comparable. However, you can define a new rectangle class that implements Comparable. The instances of this new class are comparable. Let this new class be named ComparableRectangle.

```
ComparableRectangle rectangle1 = new ComparableRectangle(4, 5);
ComparableRectangle rectangle2 = new ComparableRectangle(3, 6);
System.out.println(Max.max(rectangle1, rectangle2));
```

10

The `Cloneable` Interfaces

- 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.
- A class that implements the Cloneable interface is marked cloneable, and its objects can be cloned using the clone() method defined in the Object class.

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

11

Examples

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

```
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
```

12

Implementing Cloneable Interface

- To define a custom class that implements the Cloneable interface, the class must override the clone() method in the Object class.
- The following code defines a class named House that implements Cloneable and Comparable.

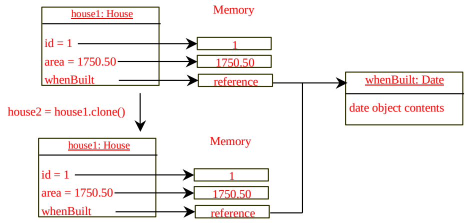
LISTING 13.11 House.java

```java
public class House implements Cloneable, Comparable<House> {
  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 int getId() {
    return id;
  }

  public double getArea() {
    return area;
  }

  public java.util.Date getWhenBuilt() {
    return whenBuilt;
  }

  @Override /** Override the protected clone method defined in
    the Object class, and strengthen its accessibility */
  public Object clone() throws CloneNotSupportedException {
    return super.clone();
  }

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

Shallow vs. Deep Copy

House house1 = new House(1, 1750.50);

House house2 = (House)house1.clone();



Interfaces vs. Abstract Classes

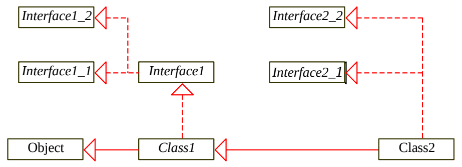
In an interface, the data must be constants; an abstract class can have all types of data.

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

	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 <u>public static final</u>	No constructors. An interface cannot be instantiated using the new operator.	All methods must be public abstract instance methods

Interfaces vs. Abstract Classes, cont.

- All classes share a single root, the <u>Object</u> 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.

Caution: conflict interfaces

In rare occasions, a class may implement two interfaces with conflict 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.

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. So their relationship should be modeled using class inheritance.
- 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.