Lecture 08 - Abstract Classes and Interfaces Part 1

CS202: Introduction to Object Oriented Programming

Victor Mejia
CSULA

Today's Topics:

- Introduction - Abstract Classes
- Case Study: Calendar and Gregorian Calendar
- Introduction - Interfaces
- The Comparable Interface

Introduction

Inheritance Hierarchy

More general
less specific

Person

Less general
more specific

Student extends
Person

Inheritance Hierarchy

More general
less specific

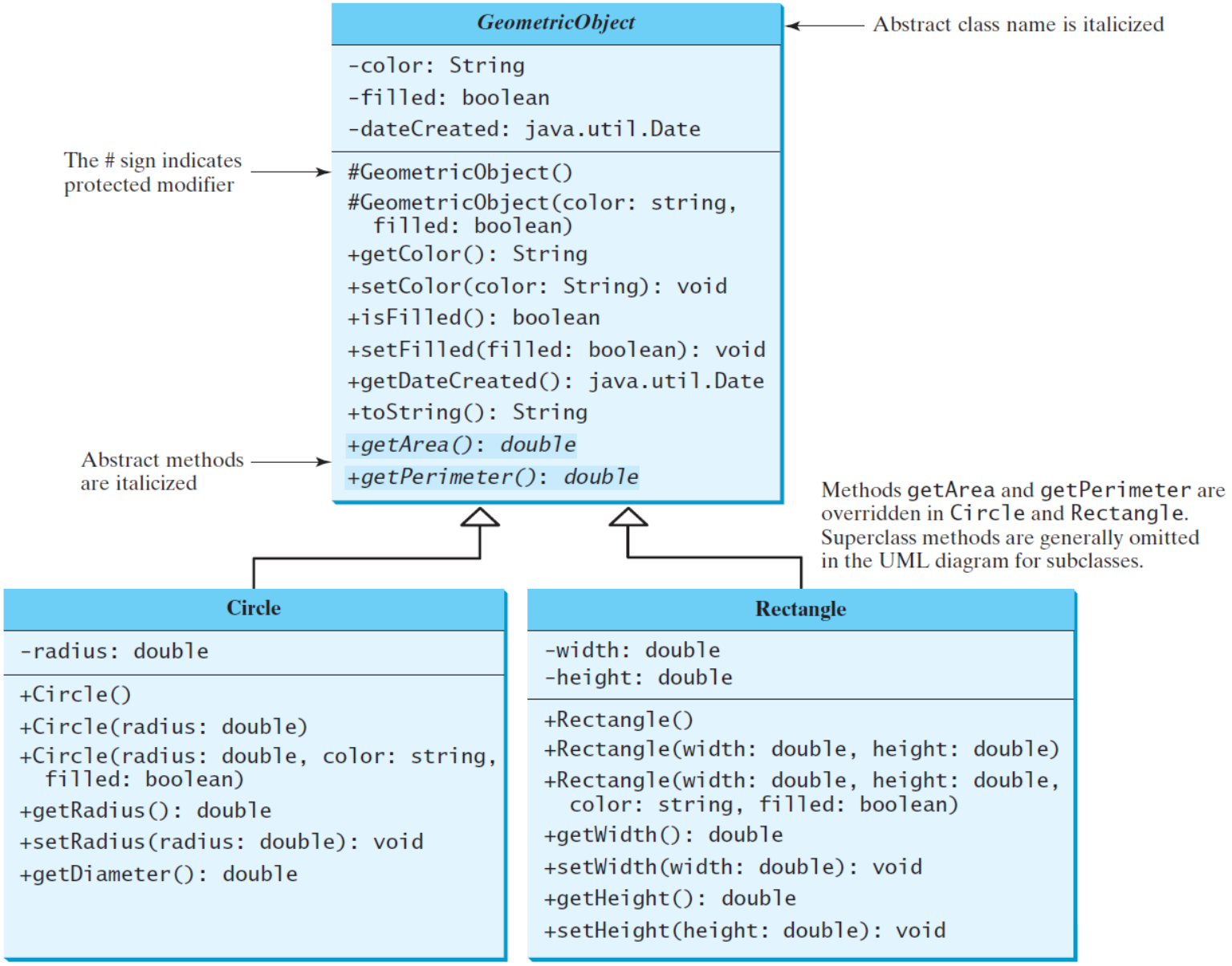
Sometimes a superclass is so abstract that it cannot be used to create any specific instances. Such a class is referred to as an abstract class.

Less general
more specific

Motivations

- In the inheritance hierarchy, classes become more specific and concrete with each new subclass.
- If you move from a subclass back up to a superclass, the classes become more general and less specific. Class design should ensure that a superclass contains common features of its subclasses. Sometimes a superclass is so abstract that it cannot be used to create any specific instances. Such a class is referred to as an abstract class

Abstract Classes and Abstract Methods



Abstract Classes and Abstract Methods

- GeometricObject.java
- Circle.java
- Rectangle.java
- TestGeometricObject.java
- TestCalendar.java

package: lecture08.abstractclasses

abstract method in abstract class

- An abstract method cannot be contained in a nonabstract class.

incorrect:

```
public class  A {
   public abstract void unfinished();
}
```

abstract method in abstract class

- An abstract method cannot be contained in a nonabstract class.

correct:

```
public abstract class  A {
   public abstract void unfinished();
}
```

abstract method in abstract 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.

object cannot be created from abstract class

- 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.

concrete method overridden to 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.

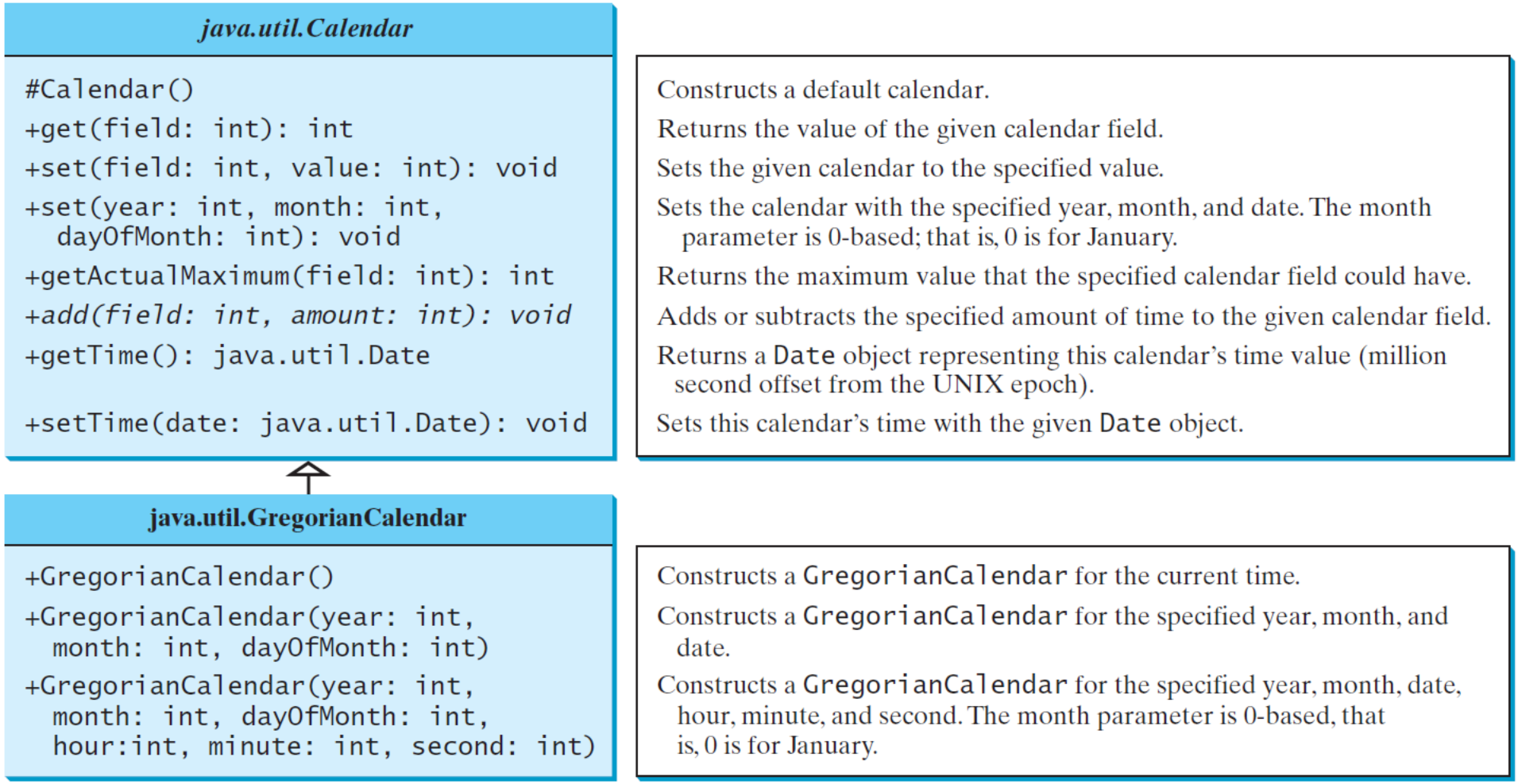
abstract class as type

- 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 GregorianCalendar Class

The Abstract Calendar Class and Its GregorianCalendar subclass



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.

Constant	*Description*
YEAR	The year of the calendar.
MONTH	The month of the calendar, with 0 for January.
DATE	The day of the calendar.
HOUR	The hour of the calendar (12-hour notation).
HOUR_OF_DAY	The hour of the calendar (24-hour notation).
MINUTE	The minute of the calendar.
SECOND	The second of the calendar.
DAY_OF_WEEK	The day number within the week, with 1 for Sunday.
DAY_OF_MONTH	Same as DATE.
DAY_OF_YEAR	The day number in the year, with 1 for the first day of the year.
WEEK_OF_MONTH	The week number within the month, with 1 for the first week.
WEEK_OF_YEAR	The week number within the year, with 1 for the first week.
AM_PM	Indicator for AM or PM (0 for AM and 1 for PM).

Interfaces

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:

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

Example:

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

Interface is a Special Class

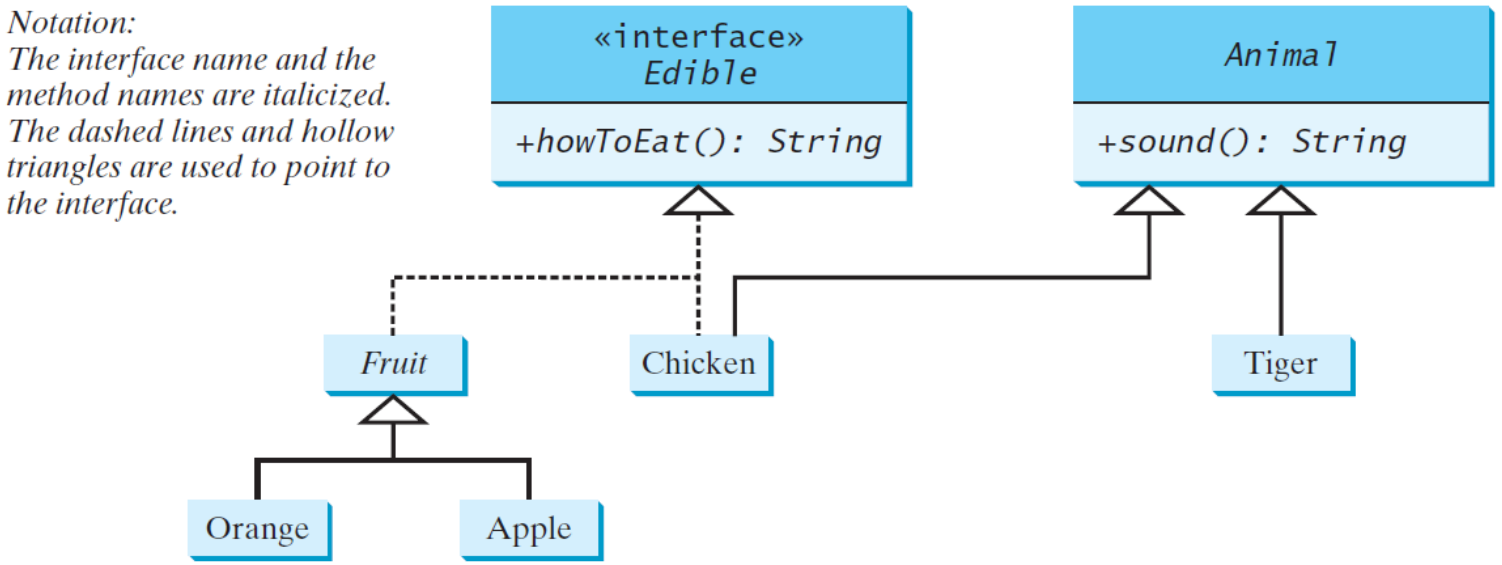
An interface is treated like a special class in Java. Each interface is compiled into a separate bytecode file, just like a regular class. Like an abstract class, you cannot create an instance from an interface using the new operator, but in most cases you can use an interface more or less the same way 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.

Edible Interface

package: lecture08.interfaces.edible

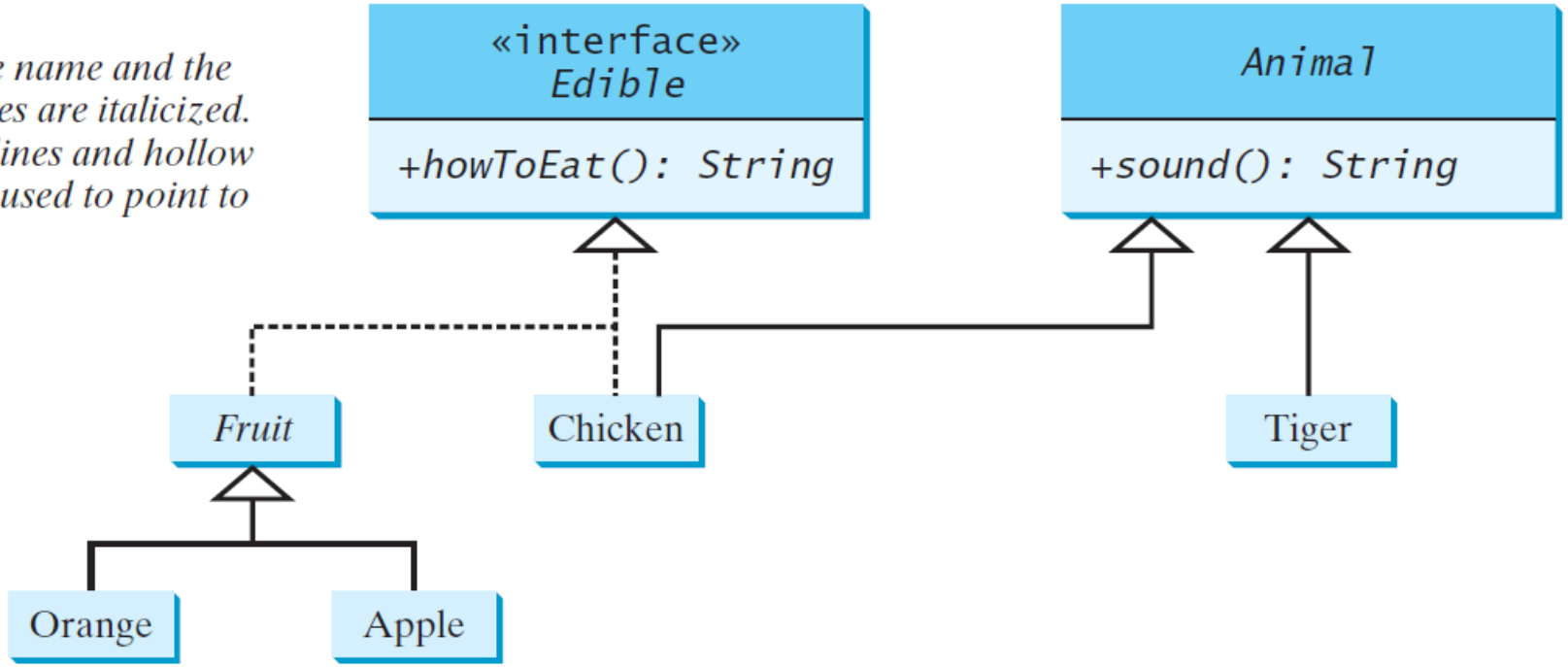
Example

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).



Example

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



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:

```
public interface T {
  public static final int K = 1;

  public abstract void p();
}
```

Equivalent

```
public interface T {
  int K = 1;

  void p();
}
```

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

The Comparable Interface

The Comparable Interface

The Comparable interface defines the compareTo method for comparing objects.

Example: The Comparable Interface

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

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

Integer and BigInteger Classes

```
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
  }
}
```

String and Date Classes

```
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

```
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 instanceof Integer
n instanceof Object
n instanceof Comparable
```

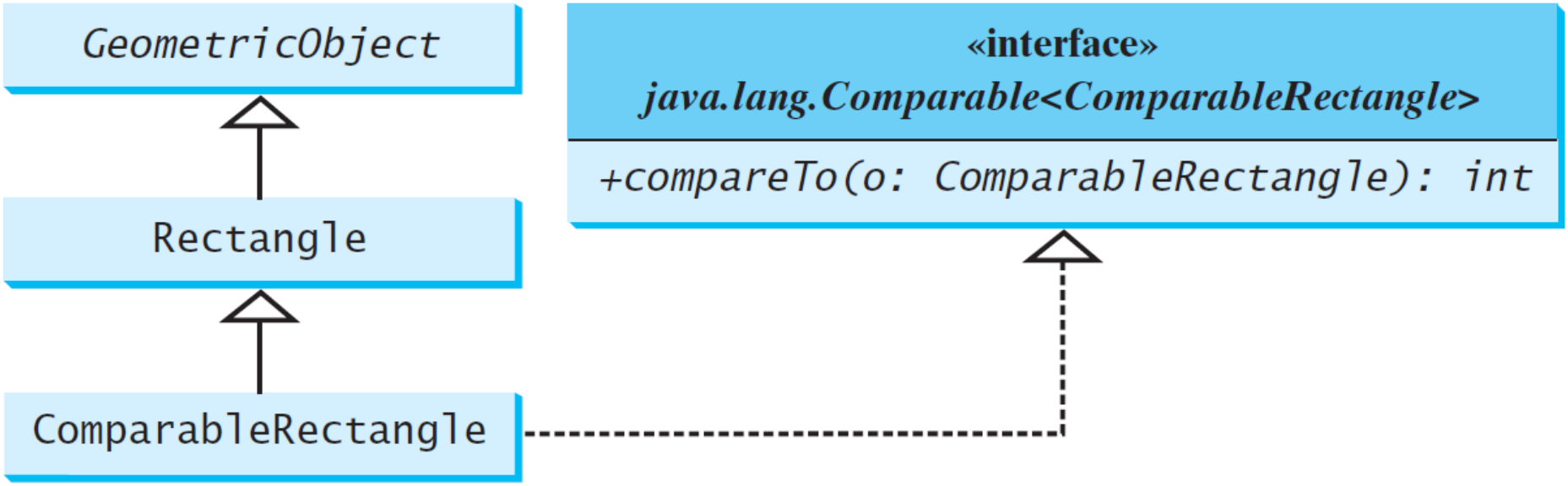
```
s instanceof String
s instanceof Object
s instanceof Comparable
```

```
d instanceof java.util.Date
d instanceof Object
d instanceof Comparable
```

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

`SortComparableObjects.java`

Defining Classes to Implement Comparable



Comparable Interfce

package: lecture08.interfaces.comparable