Chapter 13 Abstract Classes and Interfaces

为何需要抽象方法

- 前面章节中我们定义过这三个类：GeometricObject，Circle和Rectangle，其中Circle和Rectangle都有这两个方法 getArea()和getPerimeter()。
- 考虑到求面积和求周长应该是几何物体的共性，因此将这两个方法定义在GeometricObject是合适的。但是问题在于， GeometricObject只是一个抽象概念，而求面积和求周长依赖于具体的几何形状，所以这两个方法在GeometricObject中是无法具体实现的。

为何需要抽象方法

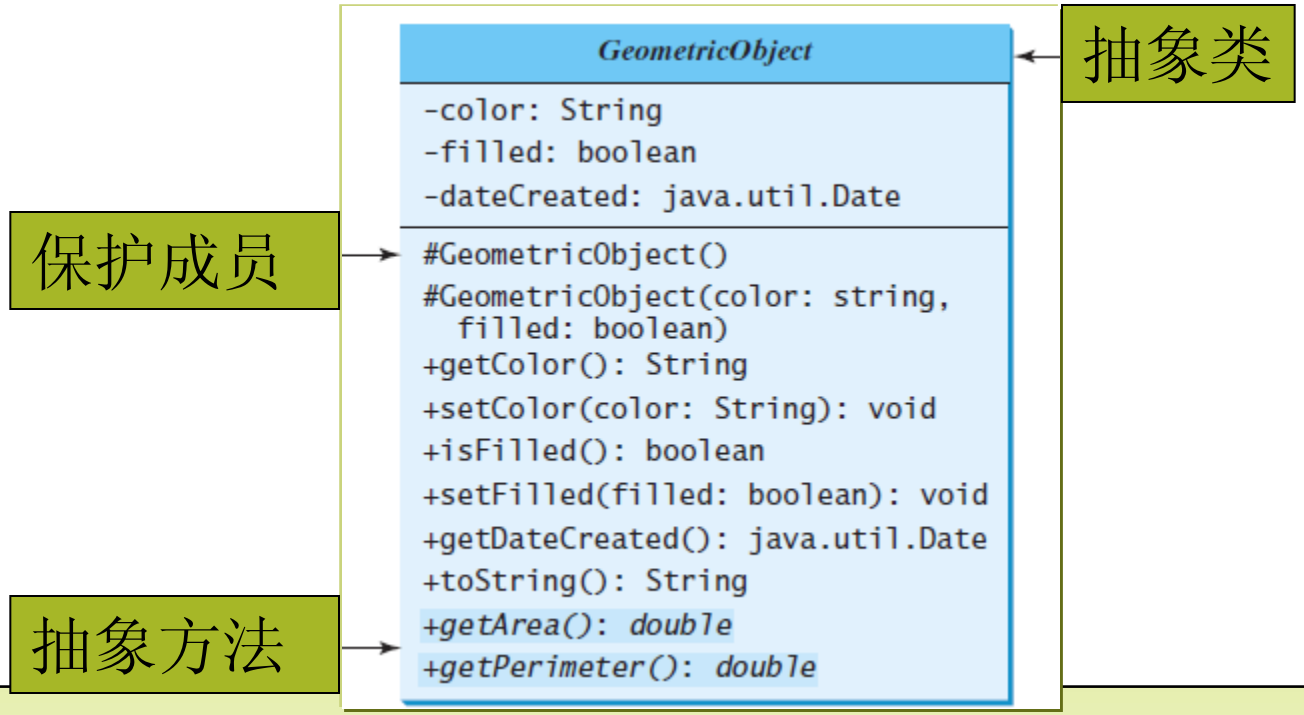
- 作为概念设计，求面积和求周长的方法在GeometricObject中必须存在；矛盾的是这两个方法又无法具体实现。作为解决方案，Java提供了抽象方法机制，允许对某些方法只定义不实现。
- 只有定义，没有具体实现的方法叫做抽象方法（*abstract methods*），这种特殊的方法需要用到 **abstract** 修饰词，例如：

```
public abstract double getArea();

public abstract double getPerimeter();
```

利用抽象方法重新定义*GeometricObject*类

- 给*GeometricObject*添加求面积和求周长两个抽象方法后，*GeometricObject*类的本身也必须被定义为抽象类。UML图中，斜体表示抽象类或抽象方法，#表示保护成员，+表示public成员。



抽象类中的抽象方法

- 抽象方法只能在抽象类中被定义。
- 如果父类定义了抽象方法，子类有两种选择：
  - 如果子类不是抽象类，那么子类必须实现父类定义的**全部抽象方法**，哪怕这个方法对它没什么用处；
  - 如果子类也是抽象类，那么子类可以**选择性实现**父类定义的抽象方法，甚至一个都不理会也行。

抽象类不能用于创建对象

- 抽象类是不能被 new 出来的，不过你依然可以定义它的构造方法，因为子类需要用到父类的构造方法。
- 例如 GeometricObject 的构造方法就会在 Circle 被new出来的时候被调用，而GeometricObject本身不能被new出来。

没有抽象方法的抽象类

- 如果一个类含有抽象方法，这个类就必须定义为抽象类；不过反之并不成立，也就是说，一个抽象类甚至可以没有定义任何抽象方法，所有的方法都是普通的方法。
- 没有抽象方法的抽象类也是不能new出来的，这种类可以作为基类被子类所继承。

抽象类的超类可以是具体类

- 当超类是具体类的时候，子类可以是抽象类。例如 Object 类就是具体类，不过它的子类 GeometricObject 就是抽象类。

具体方法可以被抽象方法所覆盖

子类可以覆盖超类中定义的具体方法，并将其改为抽象。这种情况很少见，但是至少在一个地方会用到：子类无法实现父类定义的方法。这种情况下子类只能定义为抽象类，然后用抽象方法覆盖此方法，等待子类的子类去实现。

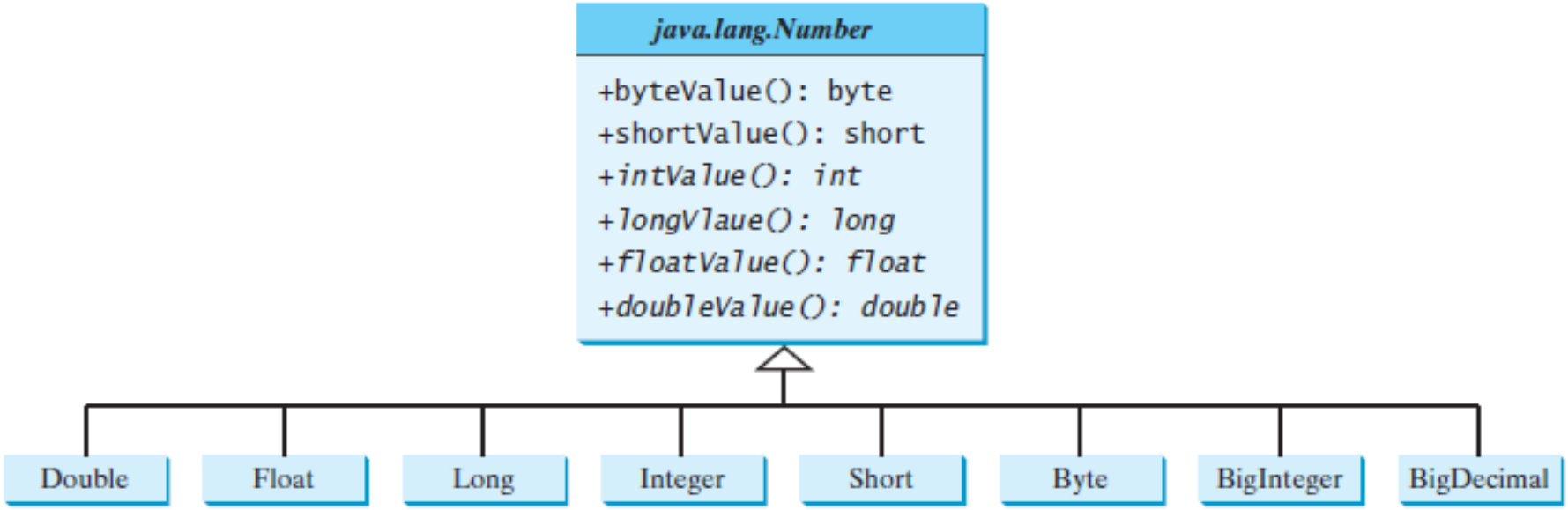
将抽象类用作数据类型

抽象类是不能new的，但是不意味着抽象类没有用，至少它可以用作数据类型。例如可以建立抽象类 GeometricObject 的数组（注意这个new是创建数组用）：

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

Number抽象类

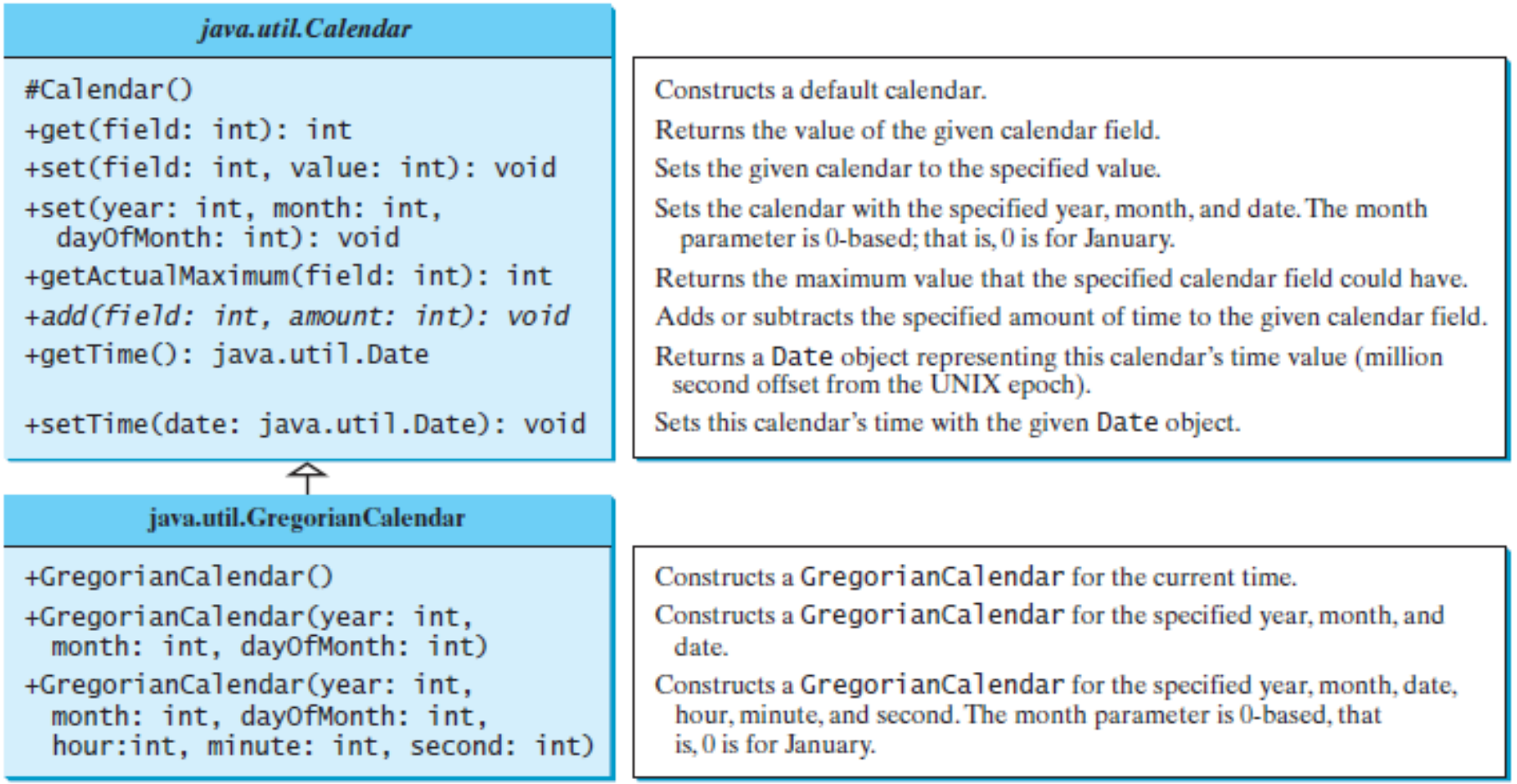
- Number抽象类定义了与数值有关的一组抽象方法，这个类是所有数值包装类以及BigInteger, BigDecimal的超类，其继承关系如图。注意类似intValue() 的方法，在Number中是无法实现的，所以只能是抽象的。



一个例子：Number虽然是抽象的，但可以当作数据类型使用。

```
1  import java.util.ArrayList;
2  import java.math.*;
3
4  public class LargestNumbers {
5    public static void main(String[] args) {
6      ArrayList<Number> list = new ArrayList<>();
7      list.add(45); // Add an integer
8      list.add(3445.53); // Add a double
9      // Add a BigInteger
10     list.add(new BigInteger("34232232344344343101"));
11     // Add a BigDecimal
12     list.add(new BigDecimal("2.0909090989091343433344343"));
13
14     System.out.println("The largest number is " +
15       getLargestNumber(list));
16   }
17
18   public static Number getLargestNumber(ArrayList<Number> list) {
19     if (list == null || list.size() == 0)
20       return null;
21
22     Number number = list.get(0);
23     for (int i = 1; i < list.size(); i++)
24       if (number.doubleValue() < list.get(i).doubleValue())
25         number = list.get(i);
26
27     return number;
28   }
29 }
```

抽象类Calendar及其子类 GregorianCalendar



Date类，Calendar类与GregorianCalendar类

- java.util.Date能表示一个时间，精度毫秒，但是不方便使用（例如不能直接获取年份）。
- java.util.Calendar是一个表示具体时间信息的抽象类，定义了从Date对象读出年份、月份、日期等具体信息的方法（但未实现）。
- java.util.GregorianCalendar是Calendar的子类，具体实现了Calendar定义的功能，所以要操作日期的话，你需要把这个类new出来。

Calendar定义的一些get方法

- Calendar类定义的get(int field)方法用于从日期抽取具体信息，field具体取值可以是：

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).

```
import java.util.*;
public class TestCalendar {
public static void main(String[] args) {
Calendar calendar = new GregorianCalendar();
System.out.println("Current time is " + new Date());
System.out.println("YEAR: " + calendar.get(Calendar.YEAR));
System.out.println("MONTH: " + calendar.get(Calendar.MONTH));
System.out.println("DATE: " + calendar.get(Calendar.DATE));
System.out.println("HOUR: " + calendar.get(Calendar.HOUR));
System.out.println("MINUTE: " +
calendar.get(Calendar.MINUTE));
System.out.println("SECOND: " +
calendar.get(Calendar.SECOND));
System.out.println("DAY_OF_WEEK: " +
calendar.get(Calendar.DAY_OF_WEEK));
System.out.println("DAY_OF_MONTH: "+
calendar.get(Calendar.DAY_OF_MONTH));
System.out.println("DAY_OF_YEAR: " +
calendar.get(Calendar.DAY_OF_YEAR));
}
}
```

注意new那行代码，calendar的类型可以用Calendar，当然使用GregorianCalendar也行。

接口

- 接口是一种特殊的抽象类，这种类只含有**常量**和**抽象方法**。很多时候，接口和一个抽象类差不多，但是接口的主要作用在于指定对象的行为。
- 例如，借助接口，你可以指定对象是可比较的，可吃的，可克隆的等等。抽象类没有这种功能。

定义一个接口

- 接口的语法如下：

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

- 例子：

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

接口是一种特殊类

- Java将接口当作一种特殊类，所以和类一样，每一个接口同样会被单独编译成字节码文件。
- 和抽象类一样，接口不能直接new出来，不过可以用作数据类型。
- 接口有两种用途：被其它类实现（implements）或被其它接口继承（extends）

接口的成员可以不写修饰符

- 由于接口中定义的属性都只能是 *public final static*，接口中定义的方法都只能是 *public abstract*，所以在接口中这些修饰符可以省略不写：

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

  public abstract void p();
}
```

Equivalent

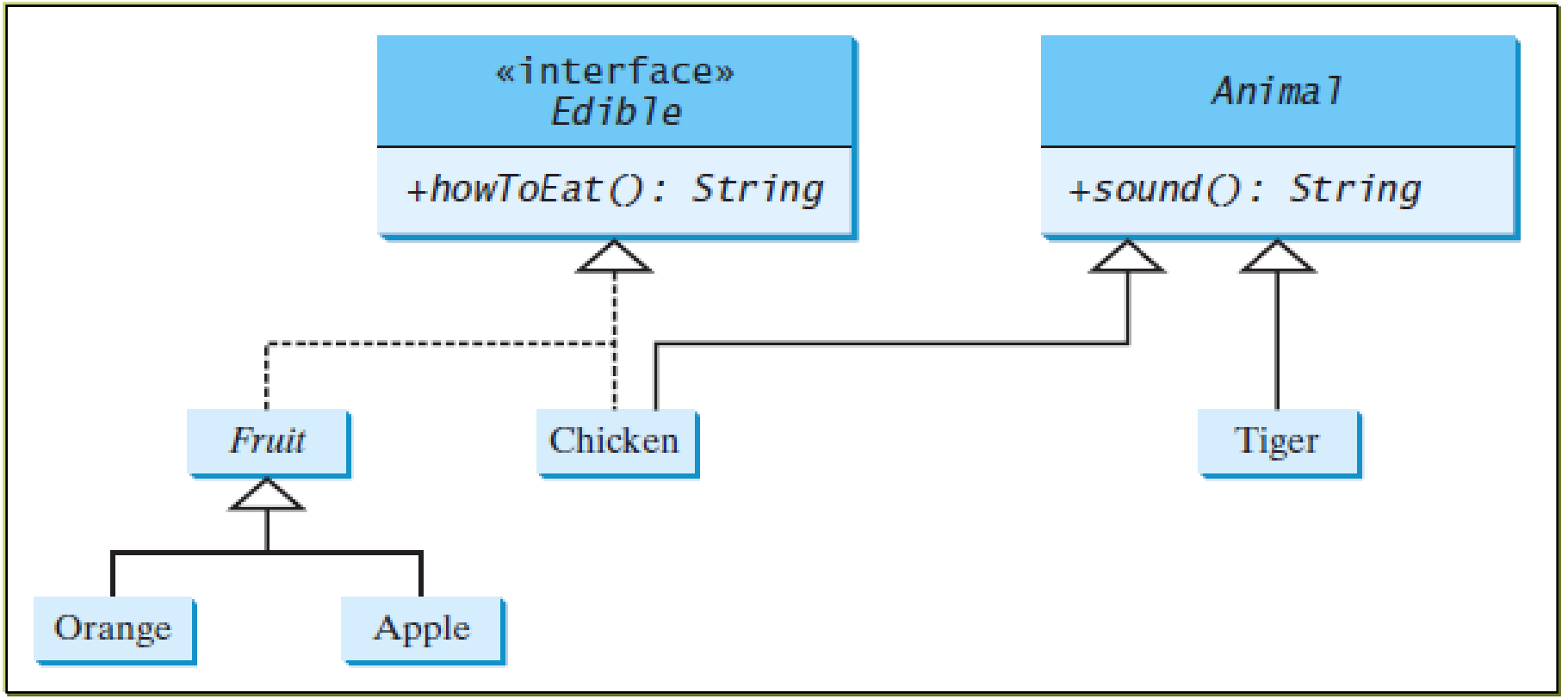
```
public interface T {
  int K = 1;

  void p();
}
```

- 接口中的常量可以使用如下语法访问：InterfaceName.CONSTANT_NAME (例如 T1.K)

一个接口的例子

- 先设计一个Edible 的接口和以下几个类：Animal，Chicken，Tiger，Fruit，Apple，Orange，它们之间的关系如下：



Edible接口和Animal、Fruit抽象类定义

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

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

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

Chicken从Animal继承，并实现了Edible接口

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

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

Tiger从Animal继承，没有实现Edible接口

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

Apple、Orange从Fruit继承，因为Fruit规定必须实现Edible接口，所以Apple、Orange被迫实现了Edible接口

```
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";
  }
}
```

测试程序

```
public class TestEdible {
  public static void main(String[] args) {
    Object[] objects = { new Tiger(), new Chicken(),
new Apple() };
    for (int i = 0; i < objects.length; i++) {
        if (objects[i] instanceof Edible)
          System.out.println(((Edible)
objects[i]).howToEat());
        if (objects[i] instanceof Animal)
          System.out.println(((Animal)
objects[i]).sound());
    }
  }
}
```

```
Tiger: RROOAARR
Chicken: Fry it
Chicken: cock-a-doodle-doo
Apple: Make apple cider
```

再来一个例子：Comparable 接口

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


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

- 约定compareTo如果返回正值，表示对象本身大于传入的对象o，0表示二者相等，负值表示对象本身小于传入的对象o。

27

数值类、字符串类和日期类显然是可以比较大小的，所以它们全都实现了Comparable 接口

```
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
  }
}
```

数值类、字符串类、日期类比较大小的例子

```
System.out.println(new Integer(3).compareTo(new Integer(5)));
System.out.println("ABC".compareTo("ABE"));
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));
```

- 输出为：

```
-1
-2
1
```

```
import java.math.*;
public class SortComparableObjects {
public static void main(String[] args) {
String[] cities = { "Savannah", "Boston", "Atlanta",
"Tampa" };
java.util.Arrays.sort(cities);
for (String city : cities)
System.out.print(city + " ");
System.out.println();

BigInteger[] hugeNumbers = { new
BigInteger("2323231092923992"),
new BigInteger("4322323239292"),
new BigInteger("54623239292") };
java.util.Arrays.sort(hugeNumbers);
for (BigInteger number : hugeNumbers)
System.out.print(number + " ");
}
}
```

这里sort之所以知道如何比较对象大小，是因为String和BigInteger实现了Comparable接口，sort会依据compareTo结果进行排序。

如果要让自定义对象也能使用sort，只要实现Comparable接口即可。

```
1  public class ComparableRectangle extends Rectangle
2      implements Comparable<ComparableRectangle> {
3    /** Construct a ComparableRectangle with specified properties */
4    public ComparableRectangle(double width, double height) {
5      super(width, height);
6    }
7
8    @Override // Implement the compareTo method defined in Comparable
9    public int compareTo(ComparableRectangle o) {
10     if (getArea() > o.getArea())
11       return 1;
12     else if (getArea() < o.getArea())
13       return -1;
14     else
15       return 0;
16   }
17
18   @Override // Implement the toString method in GeometricObject
19   public String toString() {
20     return super.toString() + " Area: " + getArea();
21   }
22 }
```

Cloneable接口

- 标识类接口（Marker Interface）是一种空接口，没有定义任何需要实现的方法。典型的是Cloneable接口，其定义如下：

```
package java.lang;
public interface Cloneable {
   //接口体为空，没有定义任何成员
}
```

- 任何一个实现了Cloneable接口的类，相当于宣称自身具有可克隆的功能，可以直接调用clone()方法克隆出一个与自身完全相同的新对象。clone()方法是在Object类定义的，与Cloneable接口没有关系。

32

Date和Calendar类

- 许多Java类实现了Cloneable接口，例如Date和Calendar，下面是它们克隆自身的例子：

```
1  Calendar calendar = new GregorianCalendar(2013, 2, 1);
2  Calendar calendar1 = calendar;
3  Calendar calendar2 = (Calendar)calendar.clone();
4  System.out.println("calendar == calendar1 is " +
5    (calendar == calendar1));
6  System.out.println("calendar == calendar2 is " +
7    (calendar == calendar2));
8  System.out.println("calendar.equals(calendar2) is " +
9    calendar.equals(calendar2));
```

输出结果如下：

```
calendar == calendar1 is true
calendar == calendar2 is false
calendar.equals(calendar2) is true
```


33

克隆数组的例子

```
1  int[] list1 = {1, 2};
2  int[] list2 = list1.clone();
3  list1[0] = 7;
4  list2[1] = 8;
5  System.out.println("list1 is " + list1[0] + ", " + list1[1]);
6  System.out.println("list2 is " + list2[0] + ", " + list2[1]);
```

displays

```
list1 is 7, 2
list2 is 1, 8
```

一个有点复杂的例子—1/4

```
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();
}
```


35

一个有点复杂的例子—2/4

```
public int getId() {
return id;
}
public double getArea() {
return area;
}
public java.util.Date getWhenBuilt() {
return whenBuilt;
}
```

一个有点复杂的例子—3/4

```
@Override
/** Override the protected clone method
defined in the Object class, and strengthen
its accessibility */
public Object clone() throws
CloneNotSupportedException {
return super.clone();
}
/* 此处覆盖了clone()方法，提升了访问权限，并调
用了父类的clone()方法。原定义如下：
protected native Object clone() throws
CloneNotSupportedException; */
```

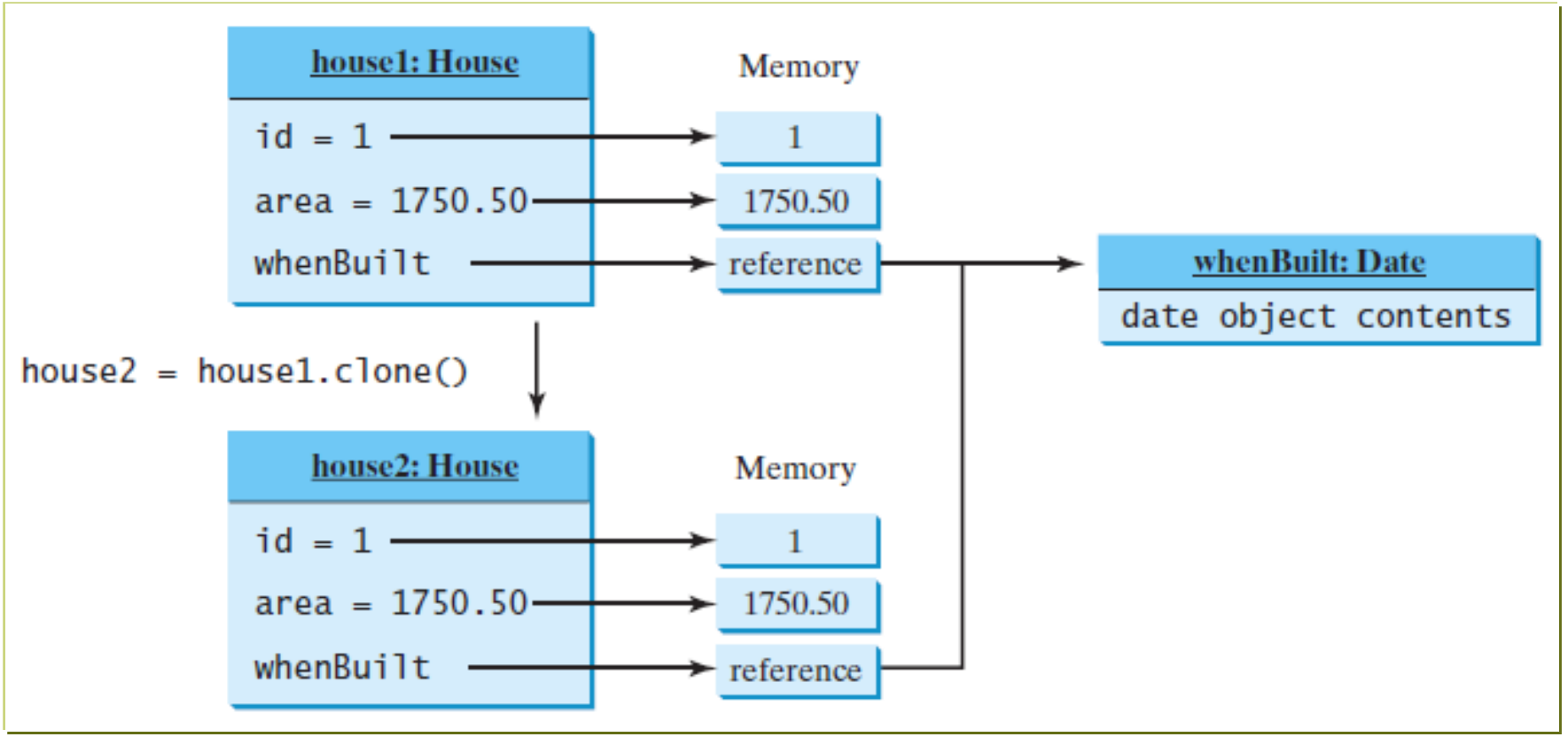
一个有点复杂的例子—3/4

```
@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;
}
}
```

浅拷贝（Shallow Copy）和深拷贝（Deep Copy）

House house1 = new House(1, 1750.50);

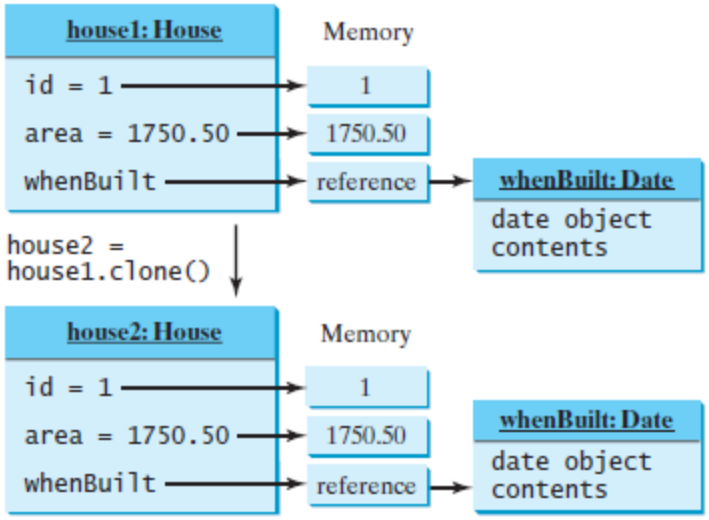
House house2 = (House)house1.clone();



解释一下

- **clone**方法会将所有的数据域从原对象复制到新对象。如果数据域是基本数据类型，直接复制原始值；如果是引用数据类型，直接复制原引用。这种做法叫做浅拷贝。
- 上例中，double型的**area**的值直接从**house1**复制到**house2**，Date型的**whenBuilt**的引用被复制到**house2**，于是：
  - **house1.whenBuilt == house2.whenBuilt**为true
  - **house1 == house2**为false

House的深拷贝版本




```
@Override
public Object clone() throws
CloneNotSupportedException {
// Perform a shallow copy
House houseClone = (House)super.clone();
// Deep copy on whenBuilt
houseClone.whenBuilt =
(java.util.Date)(whenBuilt.clone());
return houseClone;
}
```

接口和抽象类的对比—1

	成员变量	构造方法	成员方法
抽象类	无限制	构造方法由子类的构造方法链自动调用。抽象类不能使用new创建实例。	无限制
接口	必须为public static final	无构造方法。接口不能使用new创建实例。	必须为public abstract

接口和抽象类的对比－2

- Java只允许单一继承，所以抽象类只能从一个超类继承；但是Java允许一个类同时实现多个接口，例如：

```
public class NewClass extends BaseClass
Implements Interface1, . . ., InterfaceN {
. . .
}
```

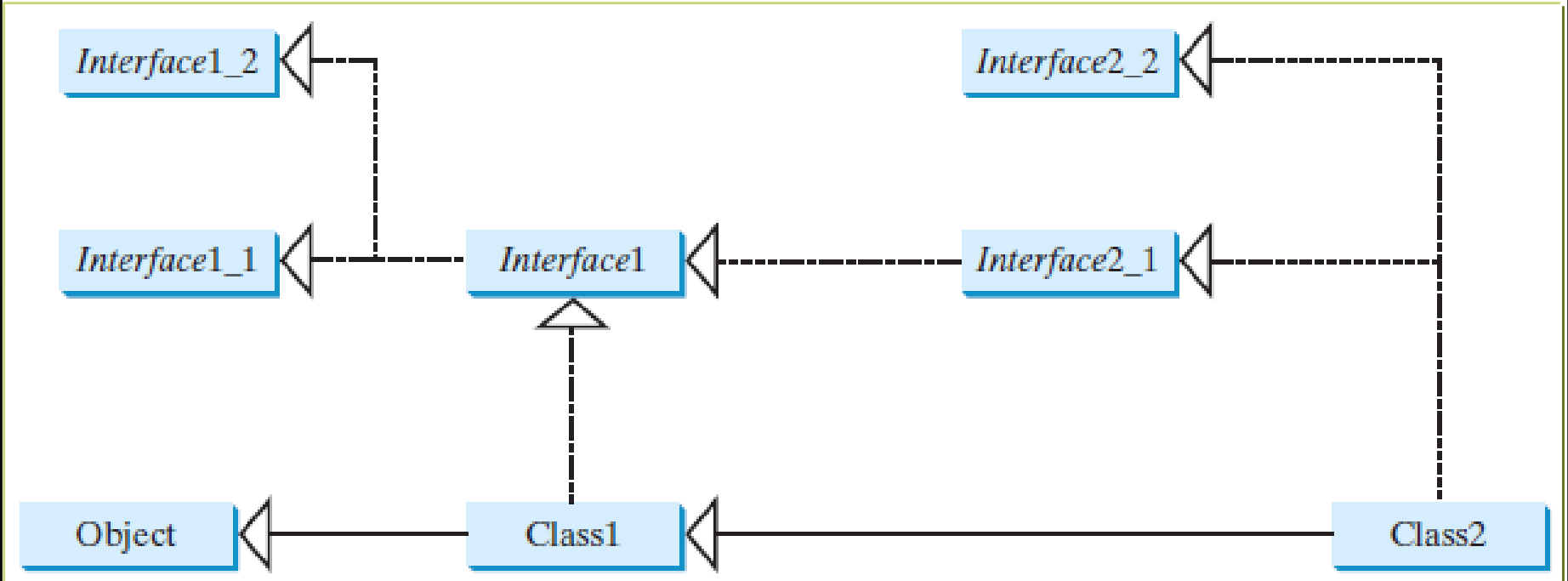
- 一个接口可以从多个接口继承，例如：

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


43

一个例子

- 如下图，假设c是Class2的实例，那么c同时也是Object, Class1, Interface1, Interface1_1, Interface1_2, Interface2_1, 和Interface2_2的实例。



44

抽象类还是接口？

- 一般而言，一种强的is-a联系能够明确表达出父子关系。例如staff是一个person，所以这种联系适合用类继承来表达；
- 一种弱的is-a联系，或者是is-kind-of联系，仅仅表明对象拥有某种属性，这种情况下用接口更加合适。例如，所有的字符串都是可比较的，所以String类实现了Comparable接口。再例如Edible，作为接口是合适的；但是用作抽象类的话，为了表明鸭子是可吃的，你只能写出下面这个难以理解的代码了：

  class Duck **extends** Edible { … } //显然Edible不适合作为超类

- 接口的另一个好处就是能够绕开Java的单一继承限制，间接实现多继承——因为一个类可以同时继承多个接口。

THE END