Interface

2301260 Programming Techniques

ผศ. ศศิภา พันธุวดีธร ภาควิชาคณิตศาสตร์และวิทยาการคอมพิวเตอร์

คณะวิทยาศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย

Interface

- Interface is used when we want to use multiple inheritance
- Interface is used when unrelated classes need to share common methods and constants
- Declaration : use interface keyword (instead of class keyword)
- Subclasses implements interface (instead of extends class)
- Multiple inheritance :

example1

interface A { … }
interface B { … }
Subclass implements A, B

example2

class A { … }
interface B { … }
Subclass extends A implements B

Interface

- Interface contains only

`public abstract` methods

and

`public static final` fields (constants)

(underline : by default = no need to write it)

- **Interface cannot be instantiated (cannot new object from interface)**
- After compiling an interface, you will get .class file as you compile general classes
- Subclasses (that `implements` the interface) must declare each method in the interface with the signature specified in the interface declaration and implements these methods
- If subclass does not implement all the methods of the interface, the subclass is an abstract class and must be declared abstract
- we can use an interface as a data type for a variable, as the result of casting

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

```
public interface T1 {
  public static final int K = 1;

  public abstract void p();
}
```

Equivalent

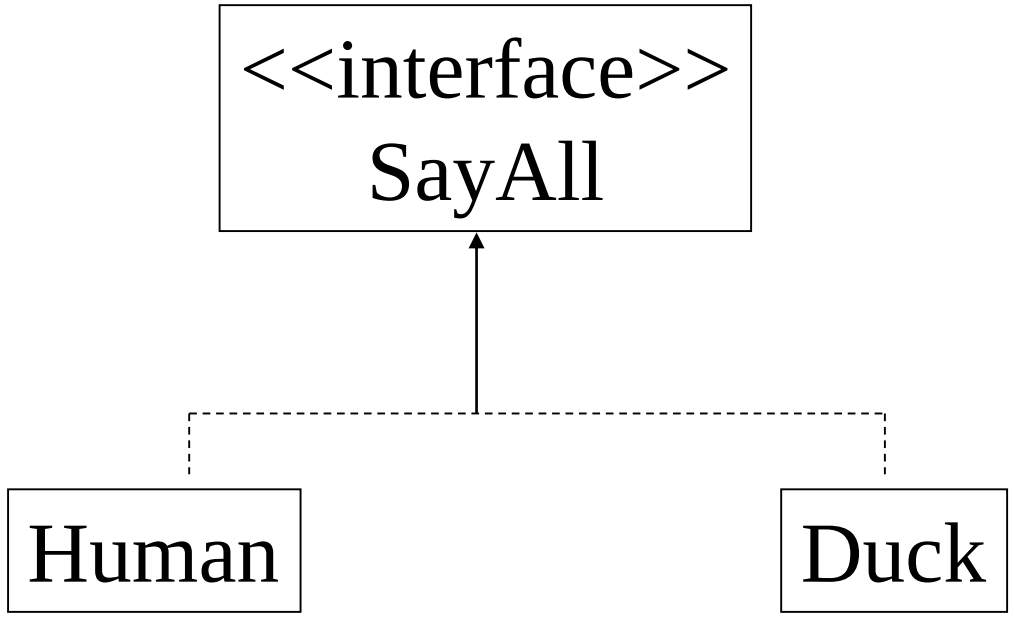
```
public interface T1 {
  int K = 1;

  void p();
}
```

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

Easy example

- Interface SayAll
- Subclass Human
- Subclass Duck
- Class Main to test objects from Human and duck



Interface SayAll (SayAll.java)

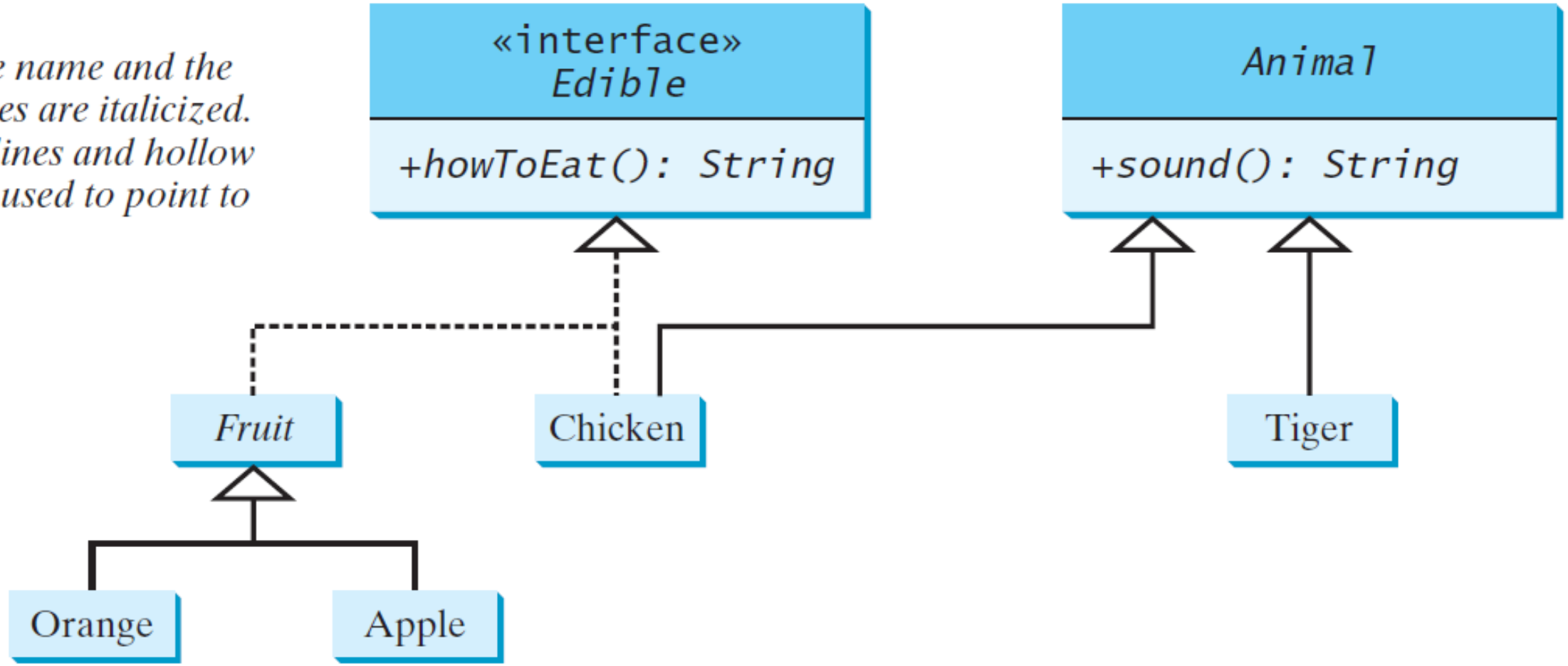
```
public interface SayAll {
    void say();                                  // public abstract void say();
}                                                // by default
public class Duck implements SayAll {
    public void say() {
        System.out.println("Gabb Gabb");
    }
}
public class Human implements SayAll {
    public void say() {
        System.out.println("Hello");
    }
}
```

Tester class to test objects

```
public class Tester {
   public static void main(String[] args) {
        Duck d = new Duck();
        d.say();
        Human sasipa = new Human();
        sasipa.say();
    }
}
```

example

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



```
public interface Edible {        /** Describe how to eat */
        String howToEat();              // public abstract String howToEat();
}
abstract class Animal {    // Animal Class
        /** Return animal sound */
        public abstract String sound();
}
class Chicken extends Animal implements Edible { // implement Edible
        public String howToEat() {
                return "Chicken Make: Fry it";
        }
        public String sound() {
                return "Chicken: cock-a-doodle-doo";
        }
}
class Tiger extends Animal {
        public String sound() {
                return "Tiger: RROOAARR";
        }
}
```

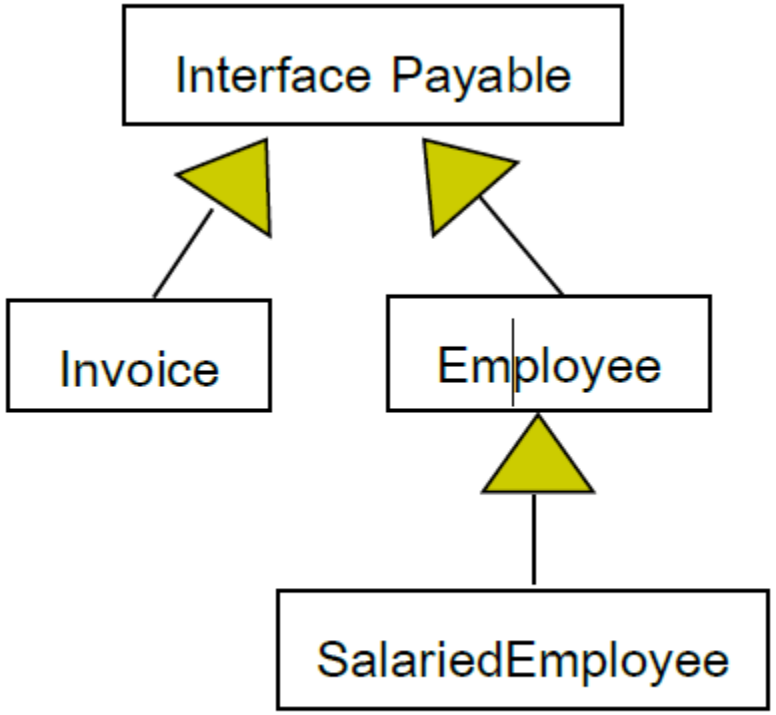
```
abstract class Fruit implements Edible {// implement Edible
        // Data fields, constructors, and methods omitted here
}
class Apple extends Fruit {          // Apple class
        public String howToEat() {
                return "Apple: Make apple cider";
        }
}
class Orange extends Fruit {  //Orange class
        public String howToEat() {
                return "Orange: Make orange juice";
        }

}

public class TestEdible {
    public static void main(String[] args) {
        Object[] objects = {new Tiger(), new Chicken(), new Apple()};  // 1. คำสั่งนี้ทำอะไร เพื่ออะไร
        for (int i = 0; i < objects.length; i++) {
                if (objects[i] instanceof Edible)                    // 2. คำสั่งนี้ทำอะไร ทำไมต้องทำแบบนี้
                        System.out.println(((Edible)objects[i]).howToEat());

                if (objects[i] instanceof Animal)
                        System.out.println(((Animal)objects[i]).sound());
        }
    }                                                                // 3. คำสั่งในส่วนที่ขีดเส้นใต้นี้ทำอะไร ทำไมต้องทำแบบนี้
}
```

ดูตัวอย่าง Interface Payable (จาก textbook ที่แต่งโดย Deitel) เพิ่มเติมในเอกสารประกอบการสอนภาษาไทยที่แจกให้



รูปที่ 8.10 ความสัมพันธ์แบบ IS-A relationship ของอินเทอร์เฟซ Payable

```
// Payable interface declaration.
public interface Payable {
   // calculate payment; no implementation
   double getPaymentAmount();
}
```

public class Invoice implements Payable ...
public abstract class Employee implements Payable ...
public class SalariedEmployee extends Employee ...

ก่อนที่จะไปอ่านโปรแกรมเพิ่มเติมอย่างละเอียด ลองตอบคำถามครูคร่าว ๆ ดูก่อน

1. Class invoice ควรทำอะไรเกี่ยวกับ Payable บ้าง
2. Class Employee ทำไมยังเป็น abstract class
3. Class SalariedEmployee ควรทำอะไรเกี่ยวกับ Payable บ้าง

Use interface for algorithm reuse

- Use interface type to make code more reusable
- In Big Java textbook, Dataset class is used to find the average and maximum of a set of numbers
- We can change the code to find the average and maximum of a set of BankAccount values or a set of Coin values
- But it is better if we use interface

ch06/dataset/DataSet.java

```
1  /**
2     Computes information about a set of data values.
3  */
4  public class DataSet
5  {
6     private double sum;
7     private double maximum;
8     private int count;
9
10    /**
11       Constructs an empty data set.
12    */
13    public DataSet()
14    {
15       sum = 0;
16       count = 0;
17       maximum = 0;
18    }
19
20    /**
21       Adds a data value to the data set
22       @param x a data value
23    */
```

ch06/dataset/DataSet.java (cont.)

```
24      public void add(double x)
25      {
26         sum = sum + x;
27         if (count == 0 || maximum < x) maximum = x;
28         count++;
29      }
30
31      /**
32         Gets the average of the added data.
33         @return the average or 0 if no data has been added
34      */
35      public double getAverage()
36      {
37         if (count == 0) return 0;
38         else return sum / count;
39      }
40
41      /**
42         Gets the largest of the added data.
43         @return the maximum or 0 if no data has been added
44      */
45      public double getMaximum()
46      {
47         return maximum;
48      }
49   }
```

Modified Dataset for BankAccount objects

```
public class DataSet {
    private double sum;
   private BankAccount maximum;
   private int count;
   ...
   public void add(BankAccount x) {
      sum = sum + x.getBalance();
      if (count == 0 || maximum.getBalance() < x.getBalance())
         maximum = x;
      count++;
   }

   public BankAccount getMaximum() {
      return maximum;
   }
}
```

Modified Dataset for Coin objects

```
public class DataSet {
   private double sum;
   private Coin maximum;
   private int count;
   ...
   public void add(Coin x) {
      sum = sum + x.getValue();
      if (count == 0 || maximum.getValue() < x.getValue())
         maximum = x;
      count++;
   }
   public Coin getMaximum() {
      return maximum;
   }
}
```

Using Interface for Algorithm Reuse

- The algorithm for the data analysis service is the same in all cases; details of measurement differ
- Classes could agree on a method `getMeasure` that obtains the measure to be used in the analysis
- We can implement a single reusable `DataSet` class whose `add` method looks like this:

```
sum = sum + x.getMeasure();
if (count == 0 || maximum.getMeasure() < x.getMeasure())
   maximum = x;
count++;
```

Using Interface for Algorithm Reuse

- What is the type of the variable `x`?
  - *`x` should refer to any class that has a `getMeasure` method*
- In Java, an **interface type** is used to specify required operations:

```
public interface Measurable
{
   double getMeasure();
}
```

- Interface declaration lists all methods that the interface type requires

UML Diagram of `DataSet` and Related Classes

- Interfaces can reduce the coupling between classes
- UML notation:
  - *Interfaces are tagged with a "stereotype" indicator «interface»*
  - *A dotted arrow with a triangular tip denotes the "is-a" relationship between a class and an interface*
  - *A dotted line with an open v-shaped arrow tip denotes the "uses" relationship or dependency*
- Note that `DataSet` is *decoupled* from `BankAccount` and `Coin`

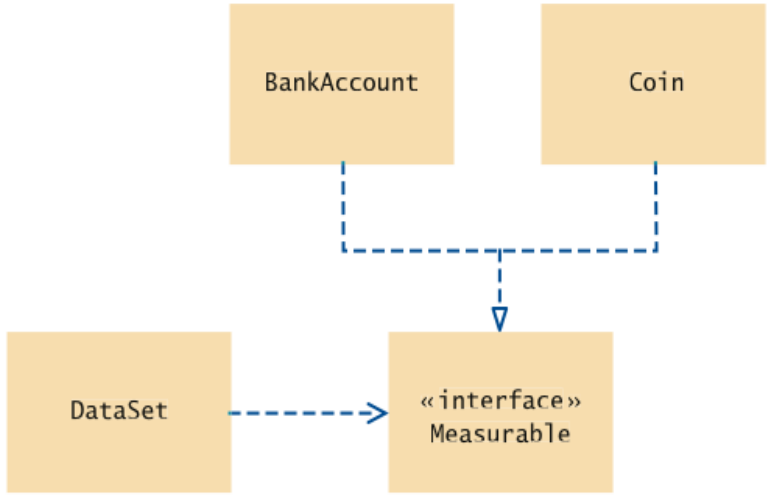


Figure 2 UML Diagram of the `DataSet` Class and the Classes that Implement the `Measurable` Interface

Generic `DataSet` for Measurable Objects

```
public class DataSet {
   private double sum;
   private Measurable maximum;
   private int count;
   ...
   public void add(Measurable x) {
      sum = sum + x.getMeasure();
      if (count == 0 || maximum.getMeasure() < x.getMeasure())
         maximum = x;
      count++;
   }

   public Measurable getMaximum() {
      return maximum;
   }
}
```

Q1 : method add() ใช้ Object x เป็นพารามิเตอร์แทน Measurable x ได้ไหม

Implementing an Interface Type

```
public class BankAccount implements Measurable {
   public double getMeasure() {
      return balance;
   }
    ...
}

public class Coin implements Measurable {
    public double getMeasure() {
       return value;
    }
    ...
}
```

ch09/measure1/DataSetTester.java

```
1  /**
2     This program tests the DataSet class.
3  */
4  public class DataSetTester
5  {
6     public static void main(String[] args)
7     {
8        DataSet bankData = new DataSet();
9
10       bankData.add(new BankAccount(0));
11       bankData.add(new BankAccount(10000));
12       bankData.add(new BankAccount(2000));
13
14       System.out.println("Average balance: " + bankData.getAverage());
15       System.out.println("Expected: 4000");
16       Measurable max = bankData.getMaximum();
17       System.out.println("Highest balance: " + max.getMeasure());
18       System.out.println("Expected: 10000");
19
```

Continued

ch09/measure1/DataSetTester.java (cont.)

```
20          DataSet coinData = new DataSet();
21
22          coinData.add(new Coin(0.25, "quarter"));
23          coinData.add(new Coin(0.1, "dime"));
24          coinData.add(new Coin(0.05, "nickel"));
25
26          System.out.println("Average coin value: " + coinData.getAverage());
27          System.out.println("Expected: 0.133");
28          max = coinData.getMaximum();
29          System.out.println("Highest coin value: " + max.getMeasure());
30          System.out.println("Expected: 0.25");
31      }
32  }

// Q2:see Assigning reference and polymorphism
// Q3:if we want to find the coin name of the max coin, what should we do?
```

ch09/measure1/DataSetTester.java (cont.)

Program Run:

```
Average balance: 4000.0
Expected: 4000
Highest balance: 10000.0
Expected: 10000
Average coin value: 0.13333333333333333
Expected: 0.133
Highest coin value: 0.25
Expected: 0.25
```

public interface Comparable in java.lang package

int compareTo(T o)

Compares this object with the specified object for order.

T - the type of objects that this object may be compared to

Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

From Employee, Salesman, Secretary in class exercise implement Comparable interface

```
public abstract class Employee implements Comparable {
    private String name;
    private int startYear;
    private double salary;
    public Employee (String name, int startYear, double salary) {
        this.name = name;
        this.startYear = startYear;
        this.salary = salary;
    }
    public String getName() {
        return name;
    }
    public int getStartYear() {
        return startYear;
    }
    public double getSalary() {
        return salary;
    }
}
```

```
public class Salesman extends Employee {
    private double sale;
    private double commRate;
    public Salesman (String name, int startYear, double salary, double sale, double
commRate) {
        super (name, startYear, salary);
        this.sale = sale;
        this.commRate = commRate;
    }
    public double getSalary() {
        return super.getSalary() + sale * commRate;
    }
    public int compareTo(Object o) {
        Salesman s = (Salesman) o;
        if (commRate == s.commRate)
            return 0;
        else if (commRate < s.commRate)
            return -1;
        else
            return 1;
    }
}
```

```
public class Secretary extends Employee {
    private int typing;
    public Secretary (String name, int startYear, double salary, int typing) {
        super (name, startYear, salary);
        this.typing = typing;
    }
    public int getTyping() {
        return typing;
    }
    public int compareTo(Object o) {
        Secretary s = (Secretary) o;
        if (typing == s.typing)
            return 0;
        else if (typing < s.typing)
            return -1;
        else
            return 1;
    }
}
```

```
public class EmployeeComparableTester {
    public static void main (String[] args) {
        Salesman s1 = new Salesman("Somying Meejai", 2005, 12500, 150000, 0.05);
        Salesman s2 = new Salesman("Somsak Pakdee", 2000, 14500, 350000, 0.08);
        Secretary c1 = new Secretary("Somjai Deejing", 2008, 20000, 60);
        Secretary c2 = new Secretary("Sompon Deejai", 2003, 25000, 60);
        if (s1.compareTo(s2) == 0)
            System.out.println("Somying and Somsak has same commRate");
        else if (s1.compareTo(s2) < 0)
            System.out.println("Somying has less commRate than Somsak");
        else
            System.out.println("Somying has greater commRate than Somsak");
        if (c1.compareTo(c2) == 0)
            System.out.println("Somjai and Sompon has same typing rate");
        else if (c1.compareTo(c2) < 0)
            System.out.println("Somjai has less typing rate than Sompon");
        else
            System.out.println("Somjai has greater typing rate than Sompon");
    }
}
```

public interface Cloneable in java.lang package

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

- this interface does *not* contain the clone method
- A class implements the Cloneable interface to indicate to the Object.clone() method that it is legal for that method to make a field-for-field copy of instances of that class.
- Invoking Object's clone method on an instance that does not implement the Cloneable interface results in the exception CloneNotSupportedException being thrown.
- By convention, classes that implement this interface should override Object.clone (which is protected) with a public method.

The `Object.clone` Method (clone() from previous chapter revisted)

- Does not systematically clone all subobjects
- Must be used with caution
- It is declared as `protected`; prevents from accidentally calling `x.clone()` if the class to which `x` belongs hasn't redefined `clone` to be `public`
- You should override the `clone` method with care

Shallow copy

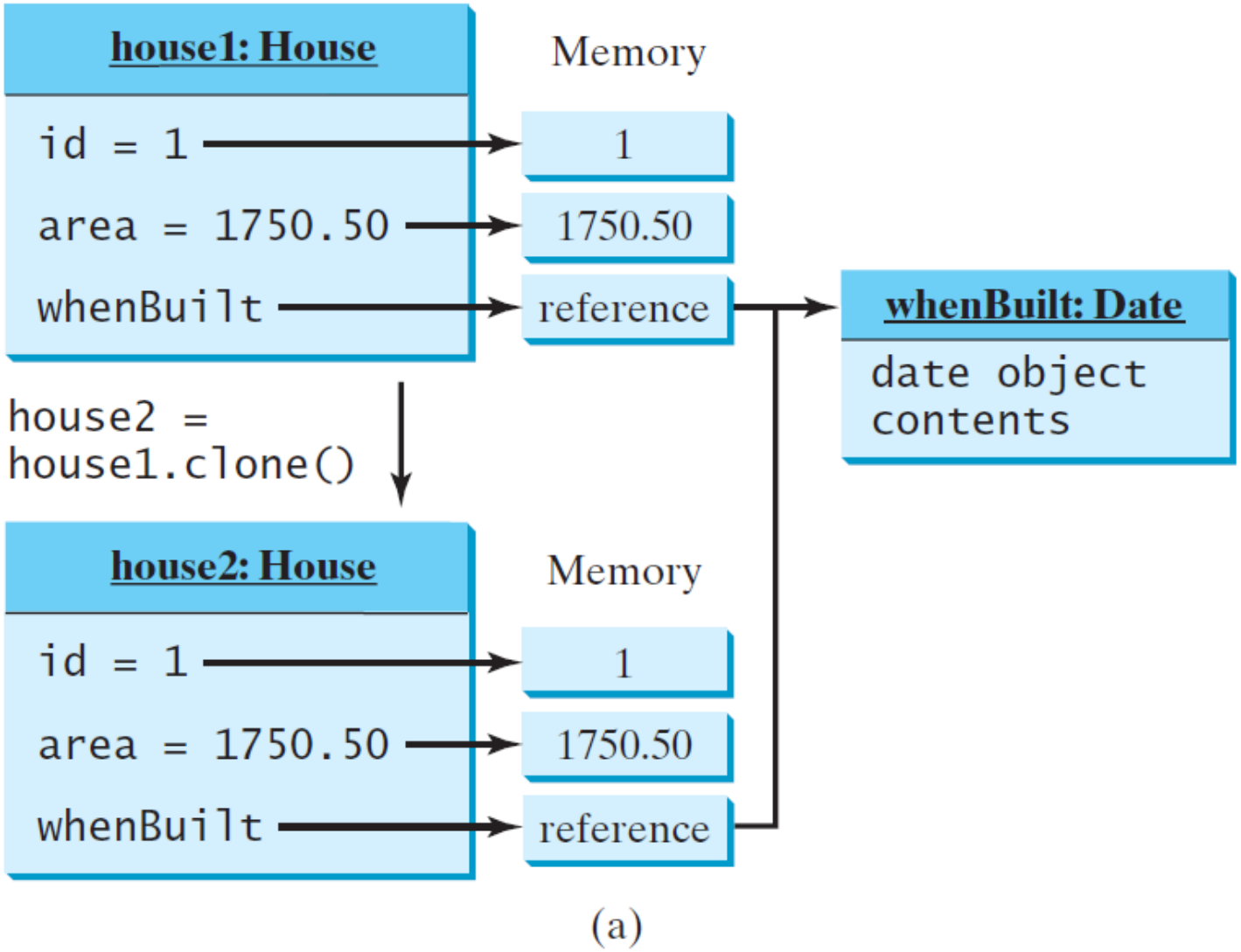
```
public class House implements Cloneable  {
    private int id;
    private double area;
    private java.util.Date whenBuilt;
    public House(int id, double area) {
        this.id = id;
        this.area = area;
        whenBuilt = new java.util.Date();
    }
    public int getId() {
        return id;
    }
    public double getArea() {
        return area;
    }
    public java.util.Date getWhenBuilt() {
        return whenBuilt;
    }
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
```

Shallow copy

```
public class HouseTester {
    public static void main(String[] args) throws CloneNotSupportedException {
        House house1 = new House(1, 1750.50);
        House house2 = (House)house1.clone();
        System.out.println("id:" + house1.getId());
        System.out.println("id:" + house1.getArea());
        System.out.println("id:" + house1.getWhenBuilt());
        System.out.println("id:" + house2.getId());
        System.out.println("id:" + house2.getArea());
        System.out.println("id:" + house2.getWhenBuilt());
        System.out.println("address of whenBuilt : " +
(house1.getWhenBuilt()==house2.getWhenBuilt()));
    }
}
```

```
run:
id:1
id:1750.5
id:Sun Mar 24 23:12:18 ICT 2019
id:1
id:1750.5
id:Sun Mar 24 23:12:18 ICT 2019
address of whenBuilt : true
```

Shallow copy



(a)

Deep copy

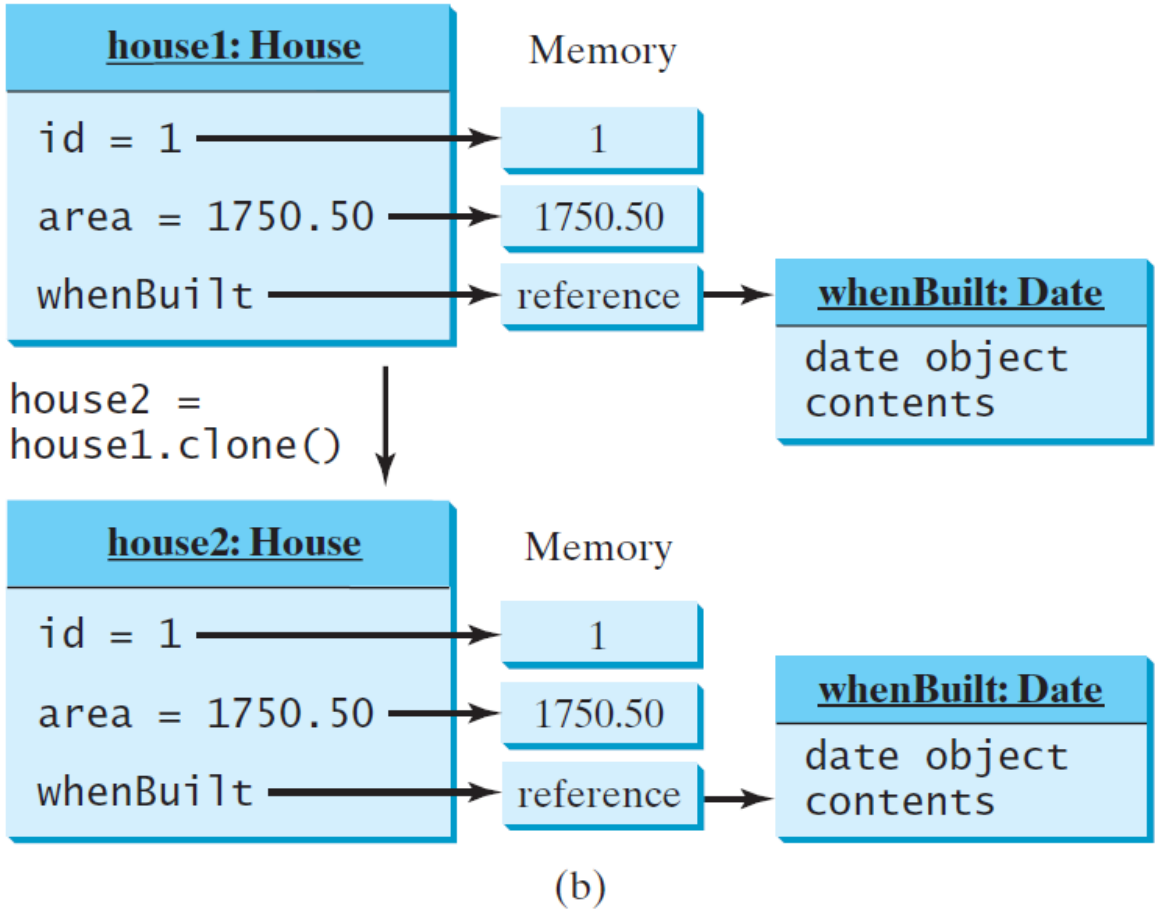
```
public class House implements Cloneable  {
    private int id;
    private double area;
    private java.util.Date whenBuilt;
    public House(int id, double area) {
        this.id = id;
        this.area = area;
        whenBuilt = new java.util.Date();
    }
    public int getId()                    {    return id;          }
    public double getArea()               {    return area;        }
    public java.util.Date getWhenBuilt()  {    return whenBuilt;   }
    public Object clone() throws CloneNotSupportedException {
        House houseClone = (House)super.clone(); // Perform a shallow copy
        houseClone.whenBuilt = (java.util.Date)(whenBuilt.clone()); // Deep copy on whenBuilt
        return houseClone;
    }
}
```

Deep copy

```
// or using try-catch
/*
    public Object clone() {
        try {
            // Perform a shallow copy
            House houseClone = (House)super.clone();
            // Deep copy on whenBuilt
            houseClone.whenBuilt = (java.util.Date)(whenBuilt.clone());
            return houseClone;
        }
        catch (CloneNotSupportedException ex) {
            return null;
    }
*/
```

Deep copy

Use the same Tester class
run:
id:1
id:1750.5
id:Sun Mar 24 23:27:07 ICT 2019
id:1
id:1750.5
id:Sun Mar 24 23:27:07 ICT 2019
address of whenBuilt : false



(b)

References

- Deitel, H.M., and Deitel, P.J., *Java How to Program*, nineth edition, Prentice Hall, 2012.
- Horstmann, C., *Big Java*, John Wiley & Sons, 2009.
- Liang, Y. D., Introduction to Java Programming, tenth edition, Pearson Education Inc, 2015.