Abstract Classes & Interfaces, More Polymorphism

Dr. Abdallah Karakra |Comp 2311 | Masri504

14/05/2023

Index

Chapter 13

- All Sections (13.1 – 13.10)

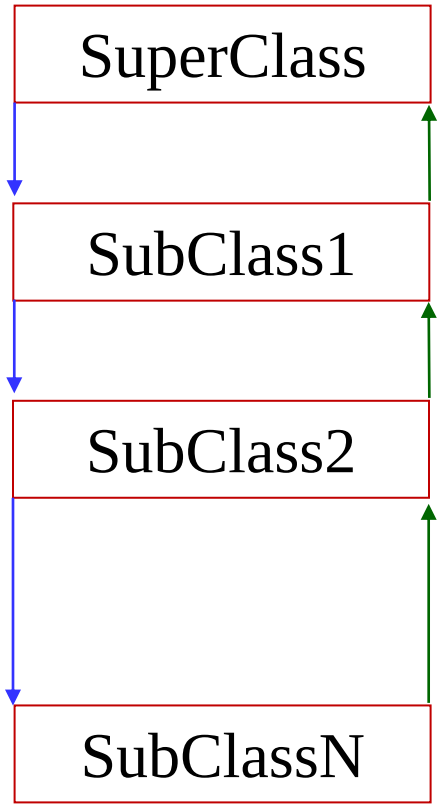
CHAPTER 13

Abstract Classes and Interfaces

Motivations

Moving from a subclass back up to a superclass, the classes become more general and less specific

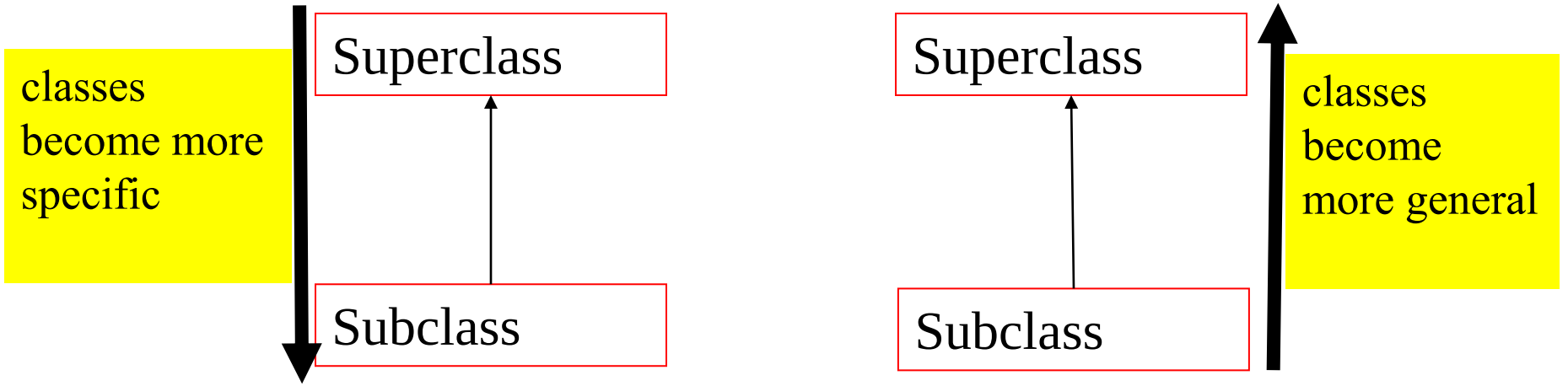
Classes become more specific and concrete with each new subclass



Sometimes, a superclass is **so general**, it **cannot** be used to create any specific instance – such a class is referred to as an abstract class

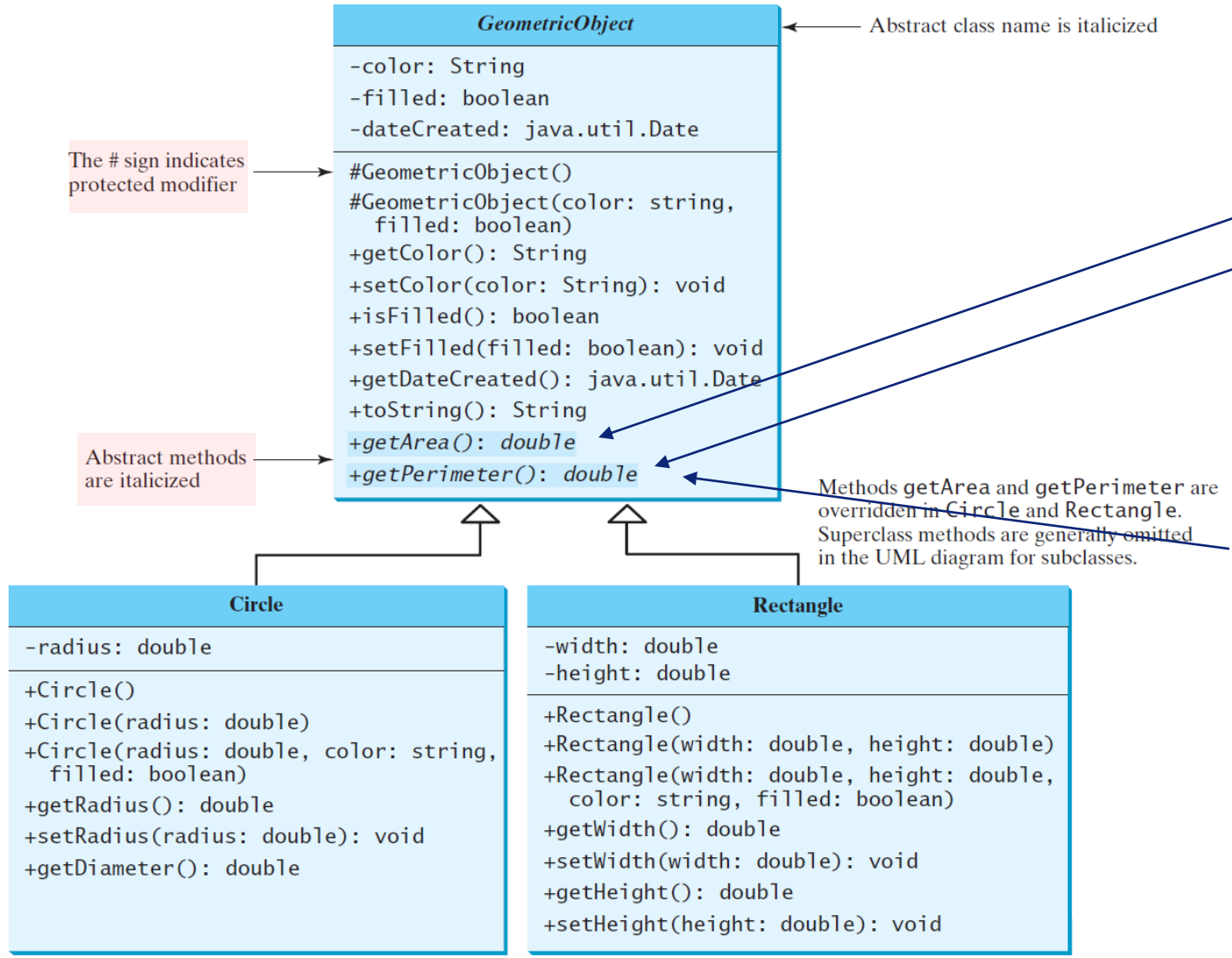
Abstract Classes

An **abstract class** cannot be used to create objects. An abstract class can contain abstract methods, which are implemented in **concrete** subclasses.



Class design should ensure that a superclass contains common features of its subclasses.

Abstract Classes and Abstract Methods



Since area and perimeter can be computed for all geometric objects, its better to define the getArea() and getPerimeter() methods in the SuperClass, GeometricObject – *designated using the abstract modifier in the class header*

However, these methods cannot be implemented in the SuperClass, GeometricObject, because their implementation depends on the specific type of geometric object – these are called abstract methods – *designated using the abstract modifier in the method header*

```java
public abstract class GeometricObject {
  private String color = "white";
  private boolean filled;
  private java.util.Date dateCreated;

  /** Construct a default geometric object */
  protected GeometricObject() {
    dateCreated = new java.util.Date();
  }

  /** Construct a geometric object with color and filled value */
  protected GeometricObject(String color, boolean filled) {
    dateCreated = new java.util.Date();
    this.color = color;
    this.filled = filled;
  }

  /** Return color */
  public String getColor() {
    return color;
  }

  /** Set a new color */
  public void setColor(String color) {
    this.color = color;
  }

  /** Return filled. Since filled is boolean,
   *  the get method is named isFilled */
  public boolean isFilled() {
    return filled;
  }

  /** Set a new filled */
  public void setFilled(boolean filled) {
    this.filled = filled;
  }

  /** Get dateCreated */
  public java.util.Date getDateCreated() {
    return dateCreated;
  }

  @Override
  public String toString() {
    return "created on " + dateCreated + "\ncolor: " + color +
      " and filled: " + filled;
  }

  /** Abstract method getArea */
  public abstract double getArea();

  /** Abstract method getPerimeter */
  public abstract double getPerimeter();
}
```

Abstract class

Abstract methods

Abstract method in abstract class

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

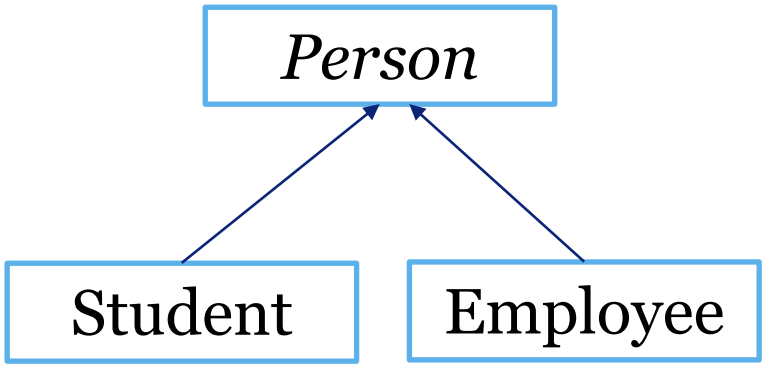
Abstract class as type

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];
```

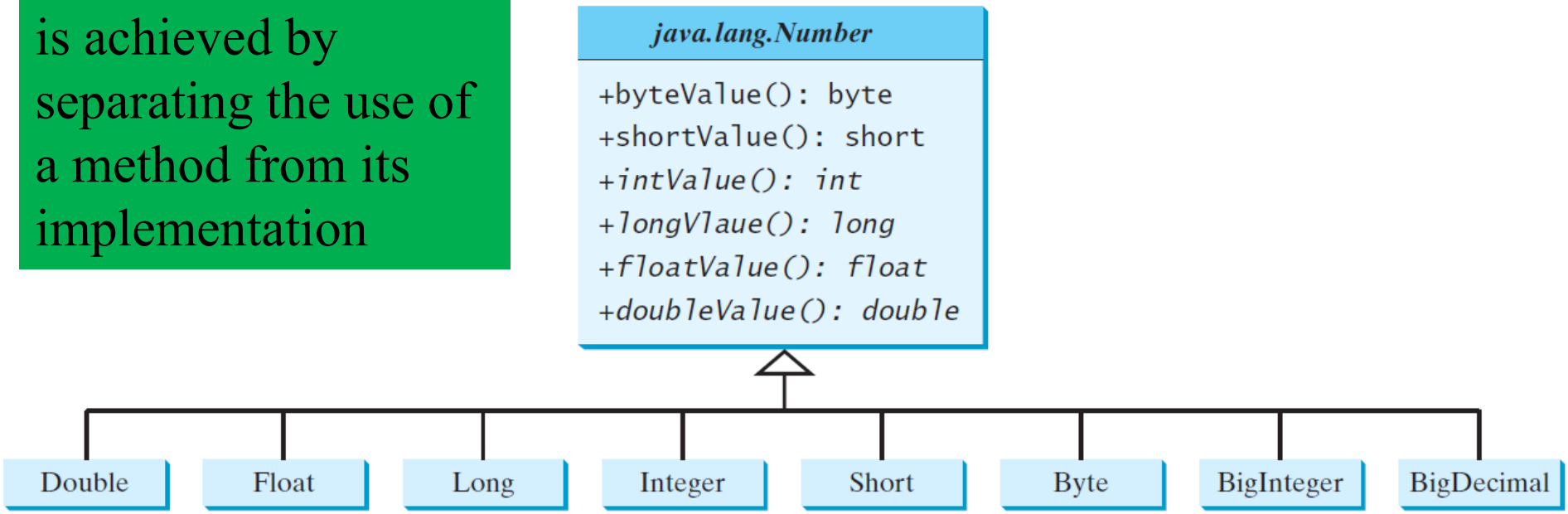
Example

```
abstract class Person {
  protected String name;
  …
  public abstract String printInfo() ;
  …
}

Class Student extends Person {
  private double avg;
  …
  public String printInfo() {
        return "Student name " +name + "  , Student avg" + avg;
  }
}

Class Employee extends Person {
  private double salary;
  …
  public String printInfo() {
        return "Employee name: " + name + " ,  Employee salary " + salary;
  }
}
```

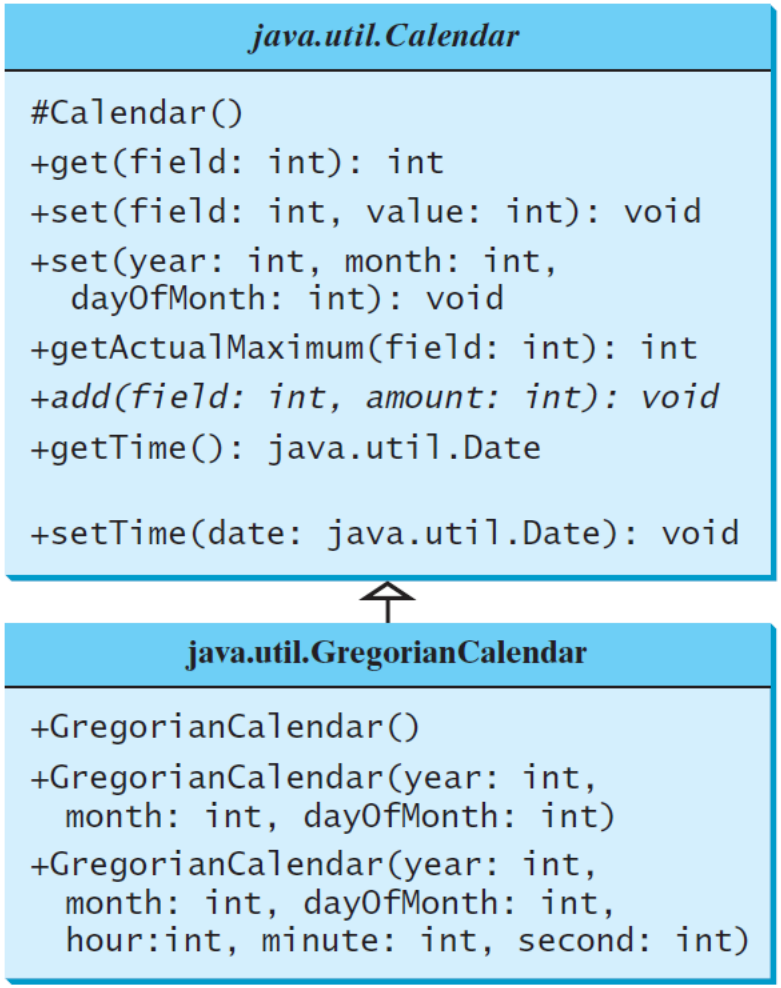


Case Study: the Abstract Number Class

Method abstraction is achieved by separating the use of a method from its implementation



The Abstract Calendar Class and Its GregorianCalendar subclass



Constructs a default calendar.

Returns the value of the given calendar field.

Sets the given calendar to the specified value.

Sets the calendar with the specified year, month, and date. The month parameter is 0-based; that is, 0 is for January.

Returns the maximum value that the specified calendar field could have.

Adds or subtracts the specified amount of time to the given calendar field.

Returns a `Date` object representing this calendar's time value (million second offset from the UNIX epoch).

Sets this calendar's time with the given `Date` object.

Constructs a `GregorianCalendar` for the current time.

Constructs a `GregorianCalendar` for the specified year, month, and date.

Constructs a `GregorianCalendar` for the specified year, month, date, hour, minute, and second. The month parameter is 0-based, that is, 0 is for January.

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).

Getting Date/Time Information from Calendar

```
public static void main(String[] args) {
  // Construct a Gregorian calendar for the current date and time
  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("HOUR_OF_DAY: " +
    calendar.get(Calendar.HOUR_OF_DAY));
  System.out.println("MINUTE: " + calendar.get(Calendar.MINUTE));
  System.out.println("SECOND: " + calendar.get(Calendar.SECOND));
```

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.

-Instead of reinventing the wheel, we can use existing classes and methods to specify common behavior for objects.

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:

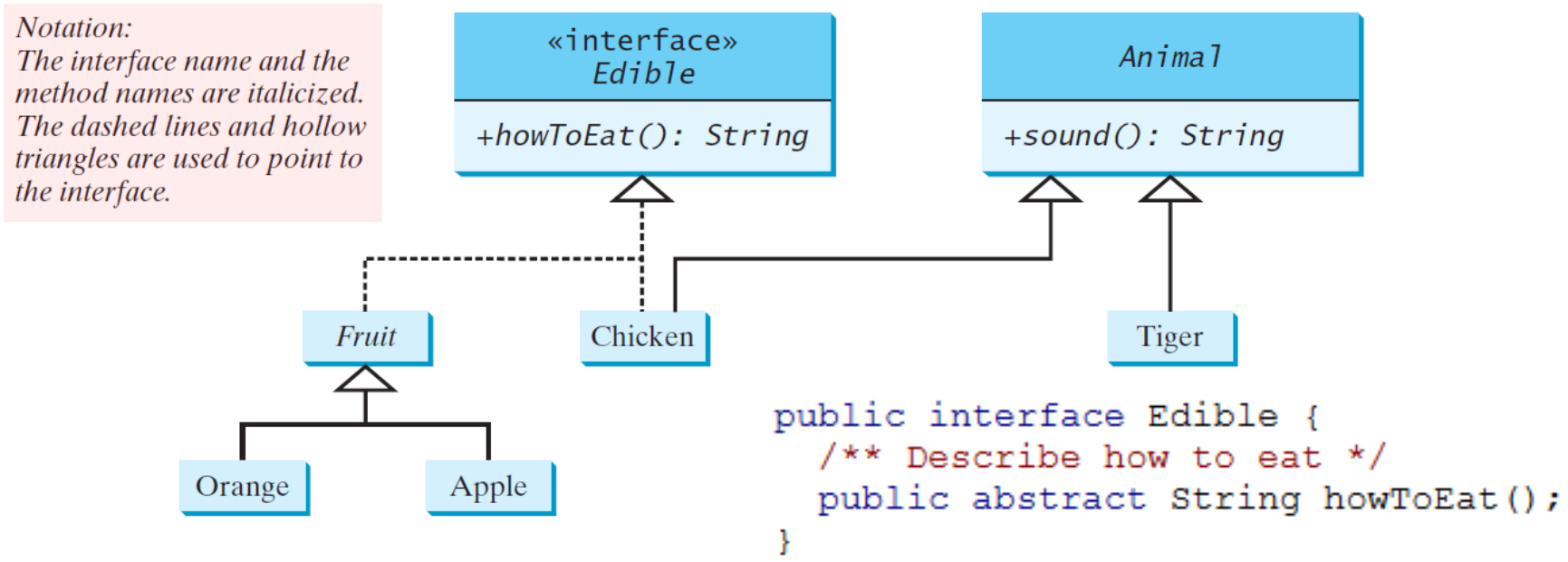
```
public interface Edible {
  /** Describe how to eat */
  public abstract String howToEat();
}
```

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.

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.



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

```java
abstract class Animal {
  private double weight;

  public double getWeight() {
    return weight;
  }

  public void setWeight(double weight) {
    this.weight = weight;
  }

  /** Return animal sound */
  public abstract String sound();
}
```

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

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

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

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

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

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

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 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)**.

Example: The Comparable Interface

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

package java.lang;
public interface Comparable<E> {
  public int compareTo(E o);
}
```

The toString, equals, and hashCode Methods

- Each wrapper class overrides the toString, equals, and hashCode methods defined in the Object class.

- Since all the numeric wrapper classes and the Character class implement the Comparable interface, **the compareTo method is implemented in these classes**.

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("ABF"));
3  java.util.Date date1 = new java.util.Date(2023, 1, 1);
4  java.util.Date date2 = new java.util.Date(2022, 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>.**

Extending Interfaces

Interfaces support **multiple** inheritance: an **interface** can extend **more** than one **interface**.

Example:

```
public interface SerializableRunnable extends
java.io.Serializable , Runnable {
                . . .
}
```

Extending Interfaces – Constants

- If a **superinterface** and a **subinterface** contain two constants with the same name, then the one belonging to the superinterface is **hidden**:

```
interface X {
        int  val  = 1;
}
interface Y extends X {
        int  val  = 2;
        int sum = val  +  X.val;
}
```

Extending Interfaces – Methods

-If a declared method in a subinterface has the same signature as an inherited method **and** the same return type, then the new declaration **overrides** the inherited method in its superinterface.

- If the **only** difference is in the return type, then there will be a **compile-time error**.

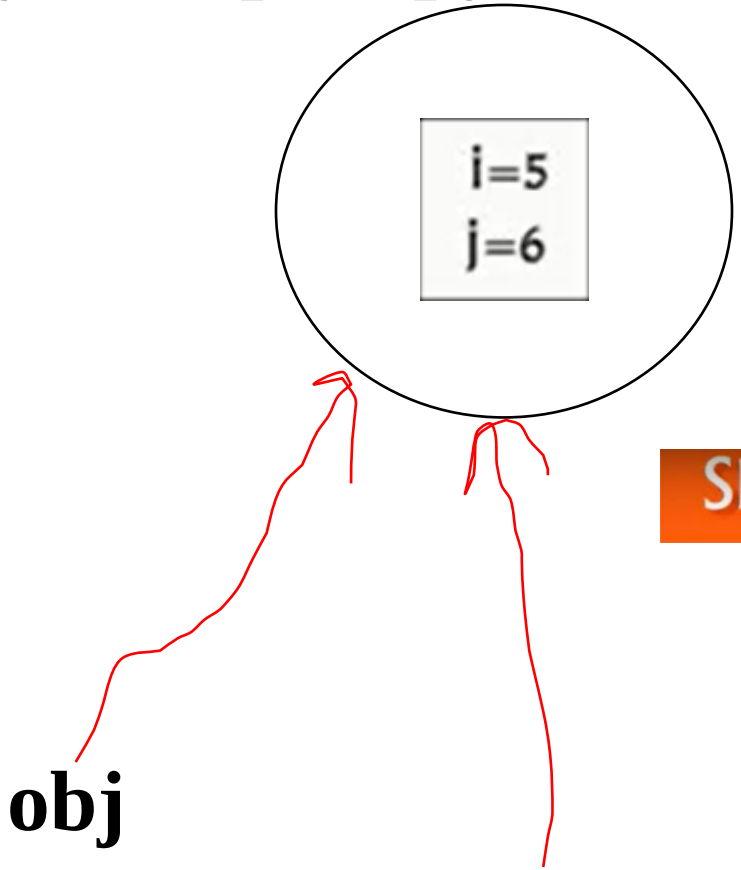
Shallow Copy , Deep Copy , and Clone

```
class A
{
   int i;
   int j;
}
```

```
A obj = new A();
obj.i=5;
obj.j=6;

A obj1= obj;
```

---> i=5 j=6

---> i=5 j=6



Shallow Copy

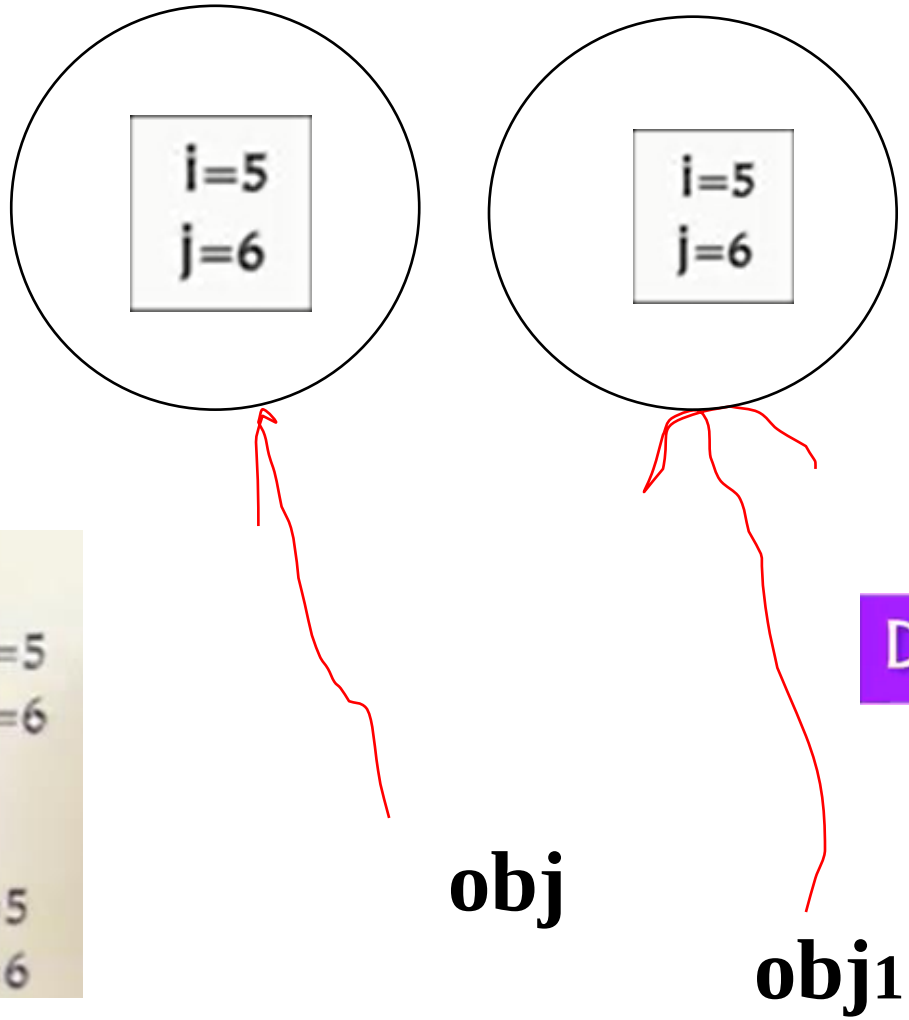
Shallow Copy , Deep Copy , and Clone

```
class A
{
   int i;
   int j;
}
```

```
A obj = new A();
obj.i=5;
obj.j=6;

A obj1= new A();
obj1,i=obj.i;
obj1.j=obj.j;
```

---> i=5 j=6

---> i=5 j=6



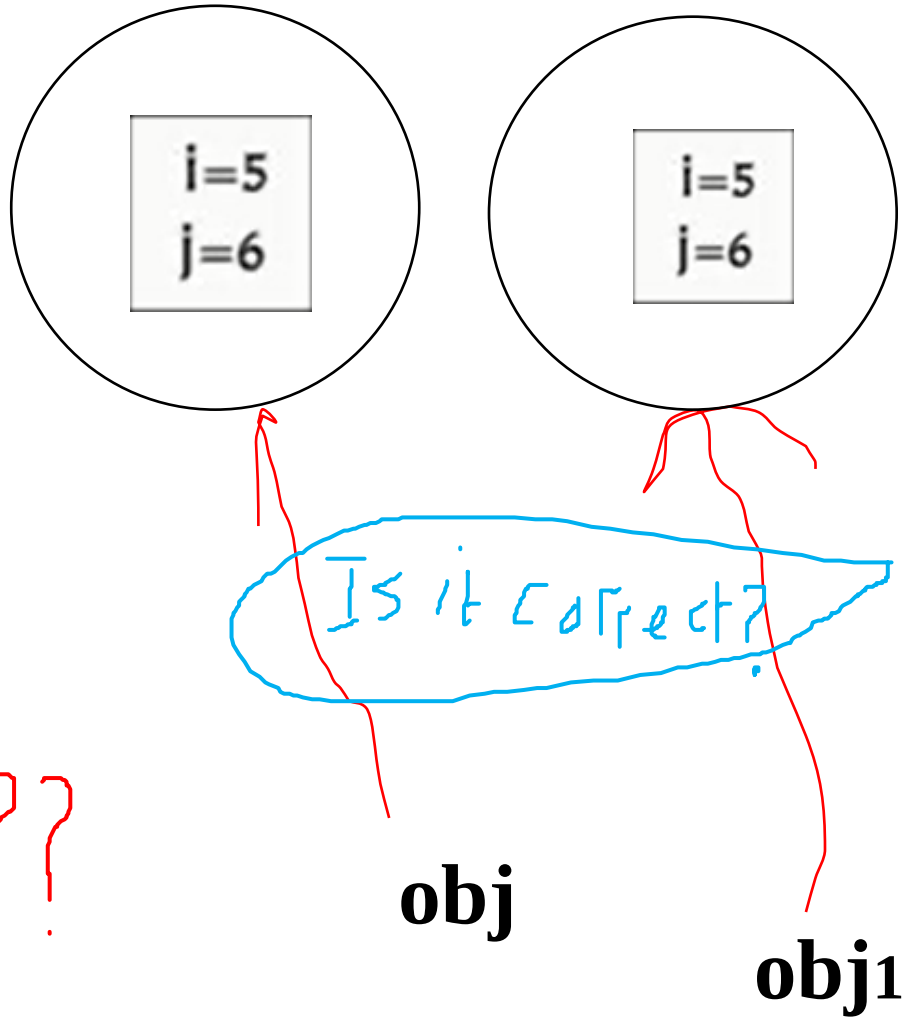
Deep Copy

What if we have 50,60, ..100 values to copy ?

Shallow Copy , Deep Copy , and Clone

```
class A
{
    int i;
    int j;
}
```

```
A obj = new A();
obj.i=5;
obj.j=6;

A obj1 =obj.clone();
```



Cloning

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 {
}
```

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
```

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.

```
public class Circle  implements Comparable<Circle>, Cloneable{

    private double radius;

    public Circle (){

        this(2.0);
    }

    public Circle(double radius){
        this.radius=radius;
    }
       //set and get methods
    public boolean equals (Object obj){
        if (obj instanceof Circle)
            return this.radius==((Circle)obj) .radius;

        return false;
    }

    public int compareTo(Circle o){

        return (int) (this.radius-o.radius);
    }

   public Object clone() {
        try{
            return super.clone();
        }catch (java.lang.CloneNotSupportedException ex){
            return null;
        }
    }
}
```

```
public class Driver{

    public static void main (String [] args){
        Circle obj= new Circle(50);
        Circle obj2= new Circle(5);

        Circle copy = (Circle)obj.clone();

        System.out.println(obj.equals(obj2));
        System.out.println(obj.compareTo(obj2));

        System.out.println(obj.compareTo(copy));
    }
}
```

```
false
45
0
```

```java
public class Circle  implements Comparable<Circle>,Cloneable {
    private double radius;

    public Circle (){
        this (3.0);
    }
    public Circle (double radius){
        this.radius=radius;
    }
    //setter and getter methods
    public boolean equals (Object obj){
        if (obj instanceof Circle )
           return this.radius==((Circle)obj).radius;
        return false;
    }
    public String toString(){
        return "circle: " + radius;
    }
    public int compareTo (Circle o){
        if (this.radius>o.radius) return 1;
        else if (this.radius<o.radius) return -1;
        else return 0;
    }
    public Object clone (){
        try{
            return super.clone();
           }
        catch (java.lang.CloneNotSupportedException ex){
            return null;
        }
    }
}
```

```java
public class Driver{
    public static void main (String [] args){
        Circle [] list = new Circle [3];
        list [0]= new Circle (4.5);
        list [1]= new Circle (2.5);
        list [2]= new Circle (5.7);

        System.out.println("Before Sorting !");
        for (Circle value: list)
          System.out.println(value.toString());

        java.util.Arrays.sort(list);

        System.out.println("After Sorting !");
        for (Circle value: list)
          System.out.println(value.toString());

    }
}
```

```
Before Sorting !
circle: 4.5
circle: 2.5
circle: 5.7
After Sorting !
circle: 2.5
circle: 4.5
circle: 5.7
```

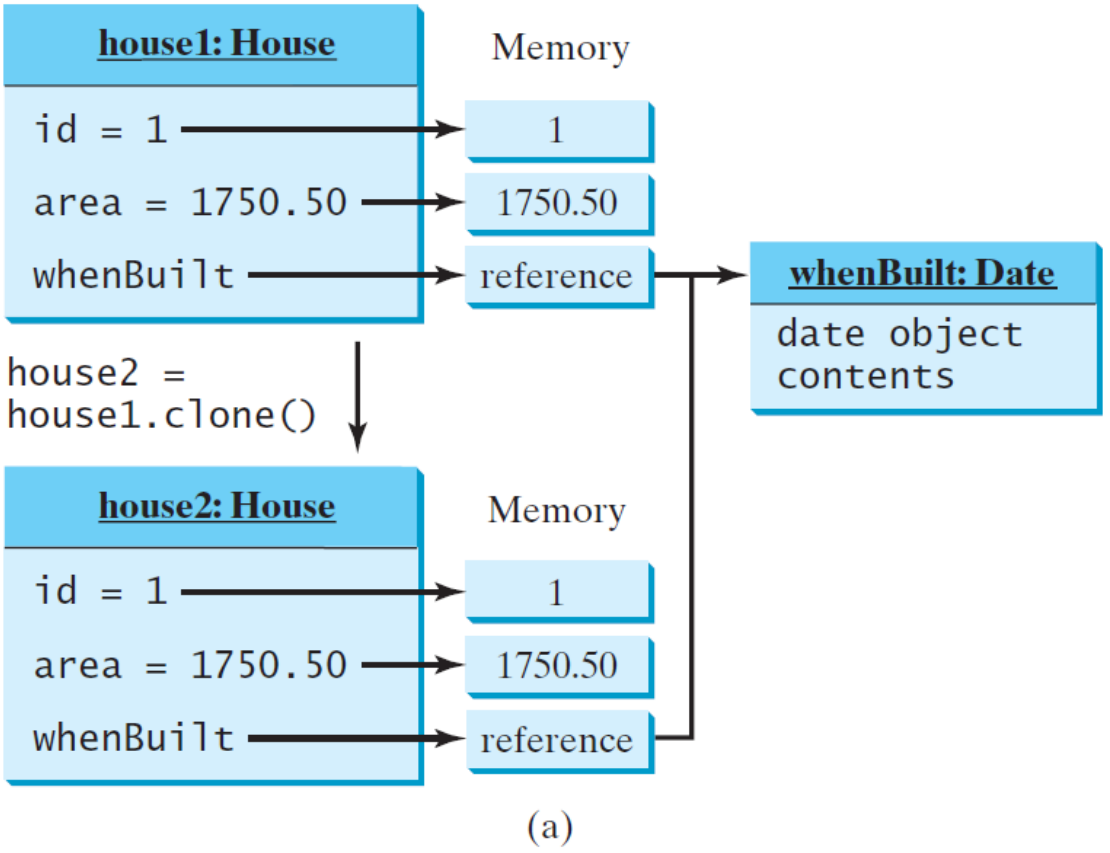
```java
public class House implements Cloneable, Comparable<House> {
  private int id;
  private double area;
  private java.util.Date whenBuilt;

  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() {
    try {
      return super.clone();
    }
    catch (CloneNotSupportedException ex) {
      return null;
    }
  }

  @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();

Shallow Copy

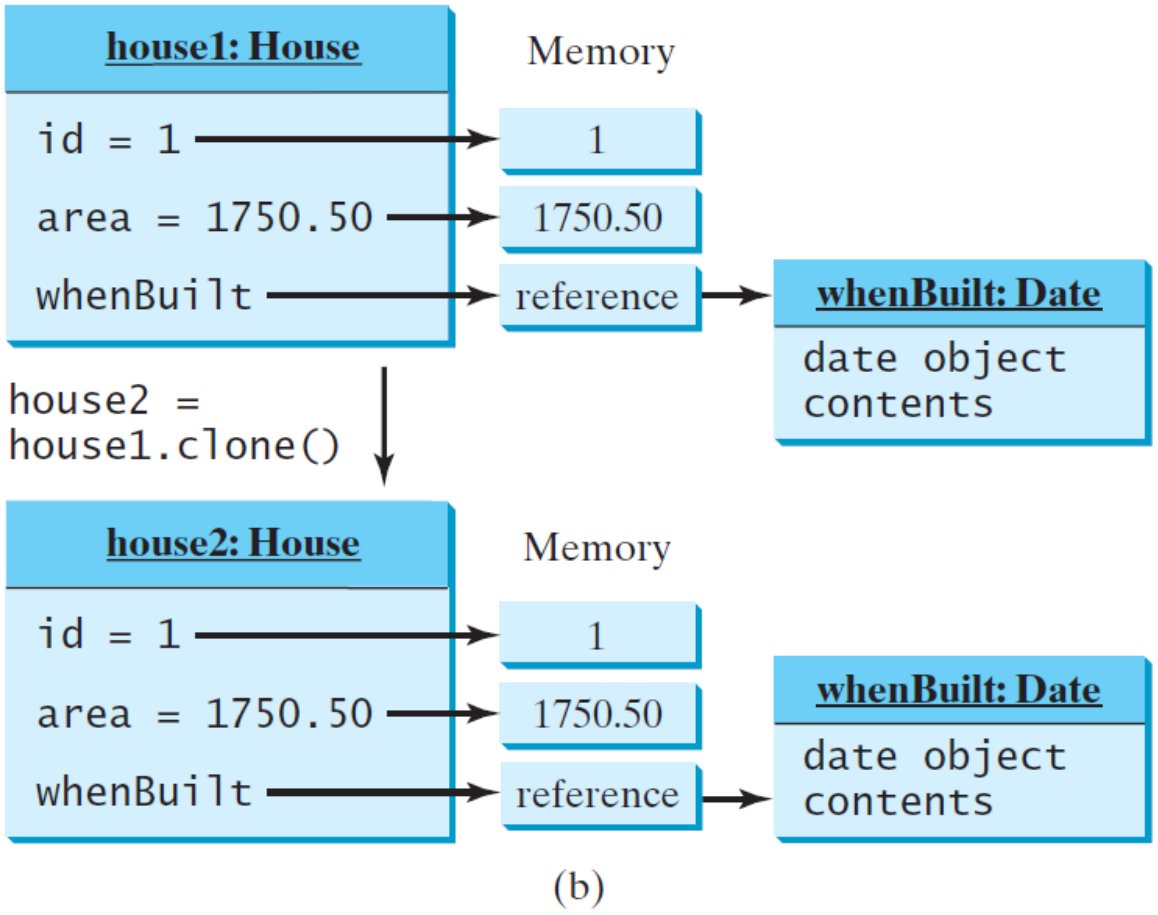


Shallow vs. Deep Copy

House house1 = new House(1, 1750.50);

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

Deep Copy



Interfaces vs. Abstract Classes

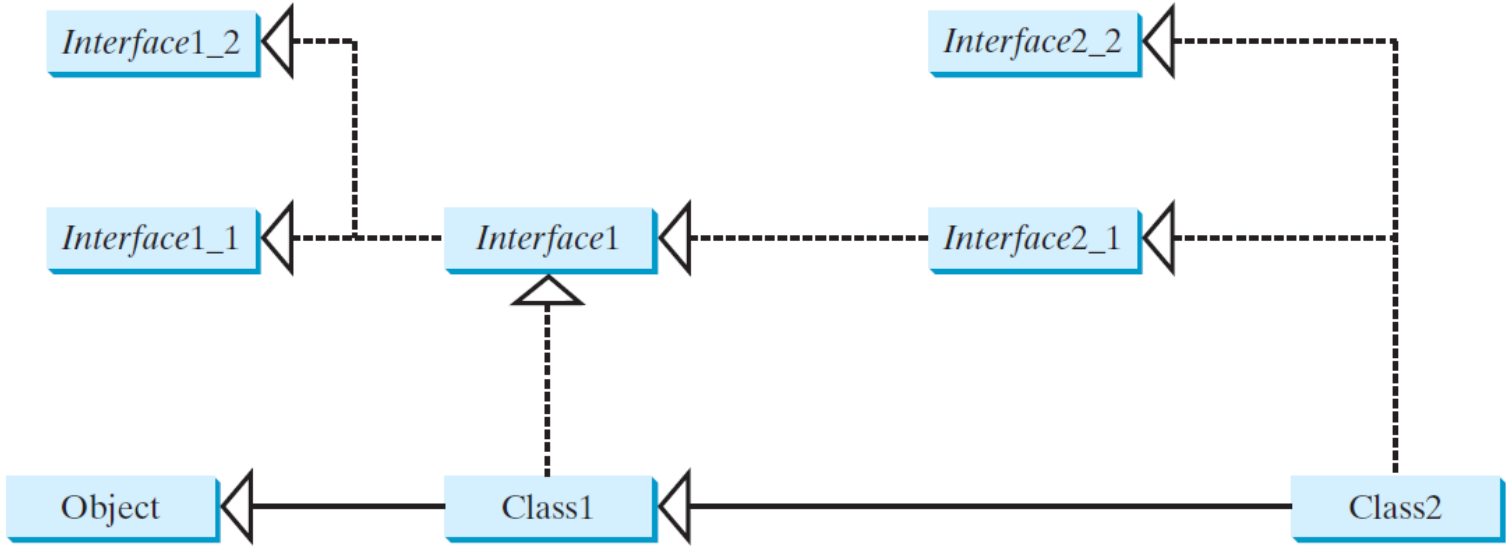
In an interface, the data must be constants; an abstract class can have all types of data.

Each method in an interface has only a signature without implementation; 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.	All methods must be public abstract instance methods

Interfaces vs. Abstract Classes, cont.

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 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.

```
interface X {
        int  val  = 1;
}
```

```
interface Y {
        int  val  = 2;
}
```

```
class  A implements X,Y{
    //reference to value is ambiguous
 //  both variable value in X and variable value in Y match
}
```

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.

Whether to use an interface or a class?

- 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.
- In the case of multiple inheritance, you have to design one as a superclass, and others as interface.

Designing a Class, cont.

- Provide a public no-arg constructor
- Override the equals method and the toString method defined in the Object class whenever possible.

Designing a Class, cont.

- Follow standard Java programming style and naming conventions.
- Choose informative names for classes, data fields, and methods.
- Always place the data declaration before the constructor.
- place constructors before methods.
- Always provide a constructor and initialize variables to avoid programming errors.

Using Visibility Modifiers

- Each class can present two contracts – one for the users of the class and one for the extenders of the class.
- Make the fields private and accessor methods public if they are intended for the users of the class.
- Make the fields or method protected if they are intended for extenders of the class.
- The contract for the extenders encompasses the contract for the users.
- The extended class may increase the visibility of an instance method from protected to public, or change its implementation, but you should never change the implementation in a way that violates that contract.

Using Visibility Modifiers, cont.

- A class should use the private modifier to hide its data from direct access by clients.
- You can use get methods and set 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.
- Some method shall be private, if it is used internally within the class.

Using the static Modifier

A property that is shared by all the instances of the class should be declared as a static property.

Abstract Classes & Interfaces, More Polymorphism