

# Lecture Notes

## Chapter #9\_c

### Abstract Classes & Interfaces

#### Abstract Classes

- parent class } → { child class  
more abstract } { more concrete, i.e., less abstract
- abstract class
  - class with an abstract modifier
  - class containing abstract methods
  - cannot create instances, i.e., objects, with the new operator

#### Listing 11.1 GeometricObject.java

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

    protected GeometricObject( ) { dateCreated = new java.util.Date( ); }

    public String getColor( ) { return color; }

    public void setColor(String color) { this.color = color; }

    public boolean isFilled( ) { return filled; }

    public void setFilled( boolean filled ) { this.filled = filled; }

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

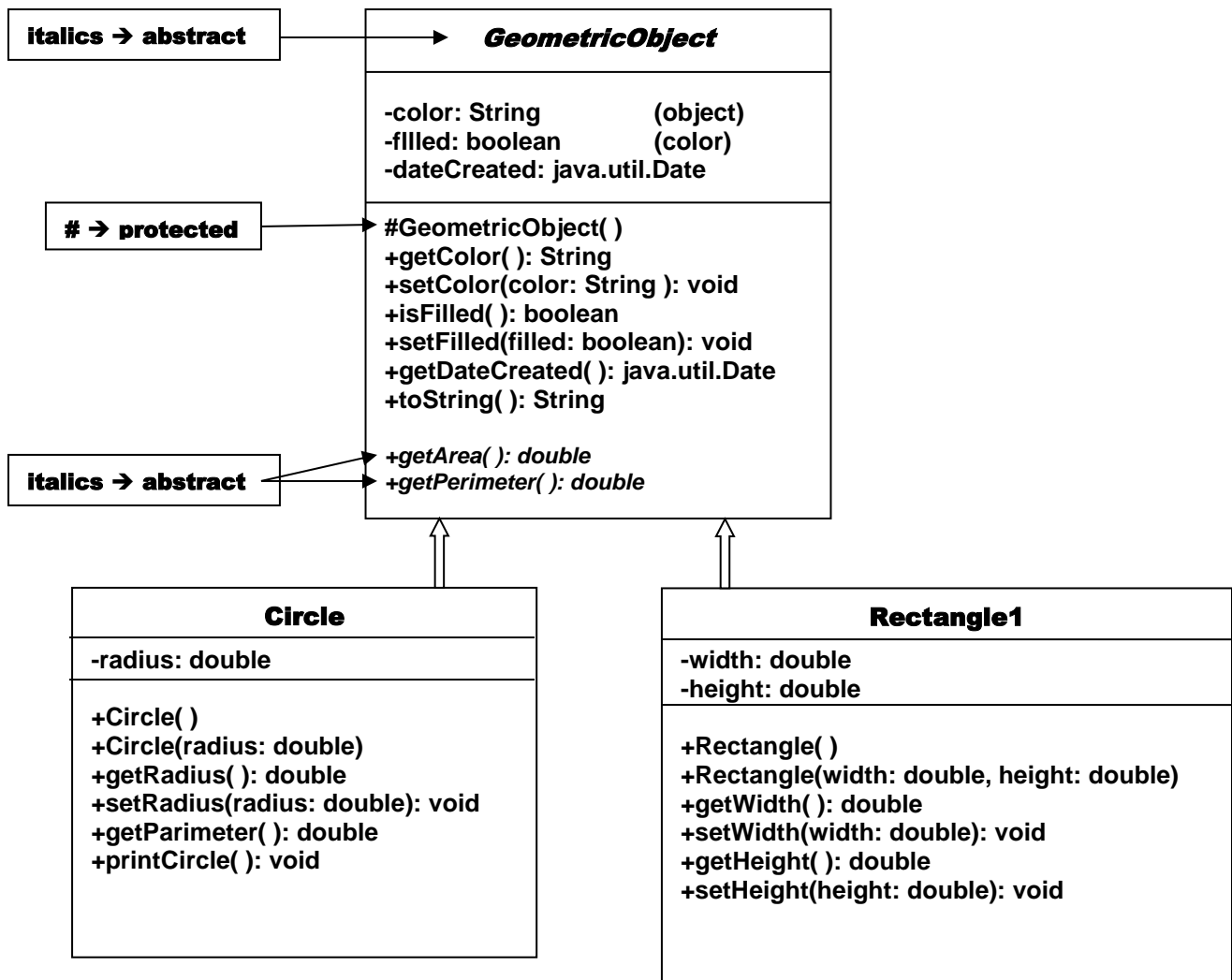
    public String toString( )
    { return "date created: " + dateCreated + "\n color: " + color + " filled: " + filled; }

    // Abstract Methods

    public abstract double getArea( );
    public abstract double getPerimeter( );
}
```

**geometricObject constructor can only be used by subclasses**

**method signatures without bodies**



Listing 11.4 TestGeometricObject.java

```

public class TestGeometricObject
{
    public static void main(String [ ] args)
    {
        GeometricObject geoObject1 = new Circle (5);
        GeometricObject geoObject2 = new Rectangle (5, 3);
        System.out.println("Object 1 ");
        displayGeometricObject(geoObject1);
        System.out.println(" and Object 2 ");
        displayGeometricObject(geoObject2);
        System.out.println("have the same area: " + equalArea(geoObject1, geoObject2));
    }

    public static boolean equalArea(GeometricObject object1, GeometricObject object2)
    { return object1.getArea( ) == object2.getArea( ); }

    public static void displayGeometricObject(GeometricObject object)
    { System.out.println("Area: " + object.getArea( ) +
        "\tPerimeter: " + object.getPerimeter( )); }
}

```

equalArea & displayGeometricObject methods are possible since the abstract class *GeometricObject* contains the abstract methods *getArea* and *getPerimeter*

Remarks

- abstract methods cannot be contained in non-abstract classes
- if a subclass of an abstract class does not implement all the abstract methods, that subclass must be declared to be abstract
- abstract methods cannot be static methods
- abstract classes cannot be instantiated by the new operator, but it can be invoked by the creation of an instantiation of a subclass object, hence it should be provided with constructors
- classes that contain abstract methods are abstract
- it is possible to declare a class which contains no abstract methods to be abstract
- a concrete superclass may have abstract subclasses, e.g., the Object class is concrete but all other classes, i.e., abstract classes, are subclasses of the Object class
- a subclass may override a concrete method from its superclass and declare it to be abstract; in this case the subclass must be declared to be abstract
- an abstract class can be used as a data type, e.g.,  

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

**Liang pages 366-368**

**java.util.Date( ) is a concrete method that represents a specific instance in time**

**java.util.Calendar is an abstract base class which contains methods for extracting detailed calendar information, e.g., year, month, etc.**

**java.util.GregorianCalendar is a concrete class which implements the abstract methods inherited from the abstract base class java.util.Calendar**

**java.lang.Number    Abstract Number class    superclass for numeric wrapper classes**

Double	Short	byteValue( )	floatValue( )
Float	Byte	shortValue( )	doubleValue( )
Long	BigInteger	intValue( )	
Integer	BigDecimal	longValue( )	

## Interfaces

- class-like construct that contains only constants and abstract methods
- used to specify common behaviors for objects
- each interface is compiled into a separate ByteCode file
- interface inheritance – multiple interfaces possible

```
public interface Edible
{
    public abstract String howToEat( );
}
```

abstract method

Listing 11.6 TestEdible.java

```
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( ));
    }
}

class Animal { }

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

class Tiger extends Animal { }

abstract class Fruit implements Edible { }

class Apple extends Fruit
{
    public String howToEat( ) { return "Apple: Make Apple Cider"; }
}

class Orange extends Fruit
{
    public String howToEat( ) { return "Orange: Make Orange Juice"; }
}
```

Remark: Since

- all data fields specified in interfaces are public final static
  - all methods specified in interfaces are public abstract
- Java allows these modifiers to be omitted

```
public interface T
{
    public static final int K = 1;
    public abstract void p( );
}
```



```
public interface T
{
    int K = 1;
    void p( );
}
```

The constant K defined in the interface T can be accessed by T.K

### Comparable Interface

- interface Comparable is defined in java.lang, e.g.,

```
package java.lang;

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

**compareTo( )** determines the order of this object with the specified object and returns a negative integer, zero, or a positive integer depending upon the rank order of the objects

Remark: Java Library objects often implement the Comparable interface to define a natural order for the objects, e.g.,

```
public class String extends Object implements Comparable { }
```

```
public class Date extends Object implements Comparable { }
```

### Generic Max Methods

```
public class Max
{
    public static Comparable max(Comparable o1, Comparable o2)
    {
        if(o1.compareTo(o2) > 0)
            return o1;
        else
            return o2;
    }
}
```

**preferable**

- simpler, more robust
- restricted to Comparable objects  
→  
compiler detects violations

```
public class Max
{
    public static Object max(Object o1, Object o2)
    {
        if(((Comparable)o1).compareTo(o2) > 0)
            return o1;
        else
            return o2;
    }
}
```

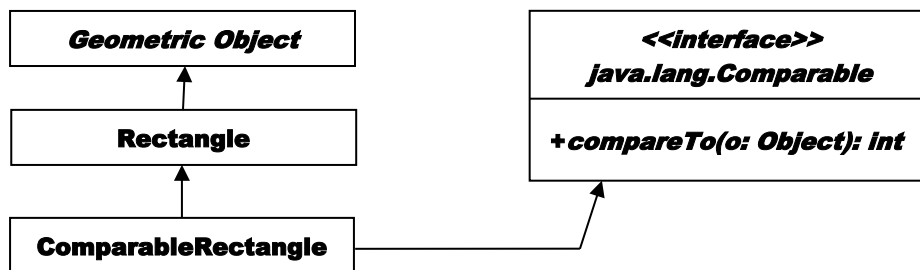
**complex, less robust method**

may be invoked by objects that have not implemented the Comparable interface hence the compiler does not detect violations; violations are detected at runtime producing the **ClassCastException**

## Listing 11.7 ComparableRectangle.java

```
public class ComparableRectangle extends Rectangle implements Comparable
{
    public class ComparableRectangle(double width, double height)
    {
        super(width, height);
    }

    public int compareTo(Object o)
    {
        if (getArea( ) > ((CompareRectangle)o).getArea( ))
            return 1;
        else if (getArea( ) < ((CompareRectangle)o).getArea( ))
            return -1;
        else
            return 0;
    }
}
```



### Remark:

- Object class contains the **equals** method
- Comparable interface provides the **compareTo** method
- Strong Recommendation: the implementation of the **compareTo** method should be consistent with the **equals** method, i.e.,  
$$o1.compareTo(o2) == 0 \iff o1.equals(o2) \text{ is true}$$

## Cloneable Interface

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

### Remarks:

- marker interface – empty interface – method has an empty body
- objects created from classes that implement the Cloneable interface can be copied by the **clone( )** method
- Java Library objects often implement the Cloneable interface, e.g., Date, Calendar, ArrayList

```
Calendar calendar = new GregorianCalendar(2003, n2, 1);  
Calendar calendarCopy = (calendar)calendar.clone( );
```

```
calendar == calendarCopy → false  
calendar.equals(calendarCopy) → true
```

two different objects with identical contents

## Implementing the Cloneable Interface

```
public class House implements Cloneable, Comparable  
{  
    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 double getId( ) { return id; }  
    public double getArea( ) { return area; }  
    public java.lang.Date getWhenBuilt( ) { return whenBuilt; }  
  
    public Object clone( ) throws CloneNotSupportedException { return super.clone( ); }  
  
    public int compareTo(Object o)  
    {  
        if (area > ((House)o).area)  
            return 1;  
        else if (area < ((House)o).area)  
            return -1;  
        else  
            return 0;  
    }  
}
```

Override the protected clone method defined in the Object class, i.e.,

**protected native Object clone( ) throws CloneNotSupportedException**

**native** → implemented in the native platform, i.e., not written in Java

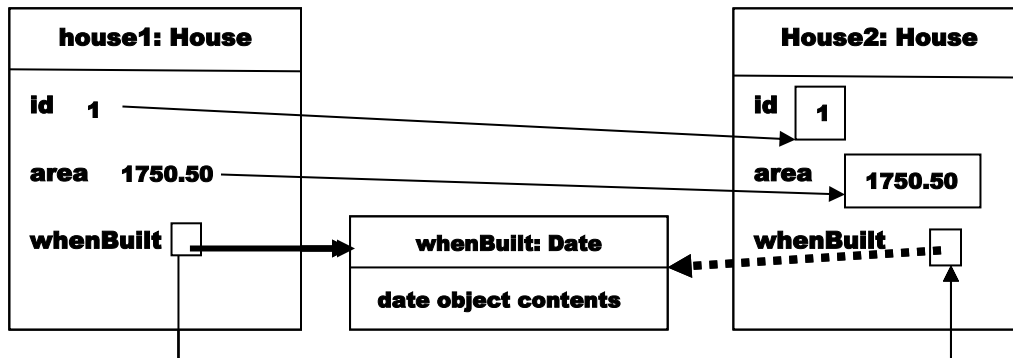
**protected** → restricts method to same package or subclass access

**override** → changes the visibility to public

## Shallow Copy

```
House house1 = new House(1, 1750.50);
House house2 = (House)house1.clone();
```

clone() copies the data value of each field in the original object to the equivalent field in the target object



## Deep Copy

override the clone method with custom cloning operations after the super.clone() constructor has been invoked

	<b>Variables</b>	<b>Constructors</b>	<b>Methods</b>
<b>Abstract Class</b>	no restrictions	invoked by subclasses via constructor chaining; cannot be invoked directly via the new operator	no restrictions
<b>Interface</b>	public static final variables	do not exist; cannot be instantiated via the new operator	public abstract instance methods

```
public class NewClass extends BaseClass implements Interface1, Interface2, ..., InterfaceN { ... }
```

## SubInterfaces

```
public interface NewInterface extends Interface1, Interface2, ..., InterfaceN { ... }
```

**Class Names -- Nouns**

**Interface Names -- Nouns or Adjectives**