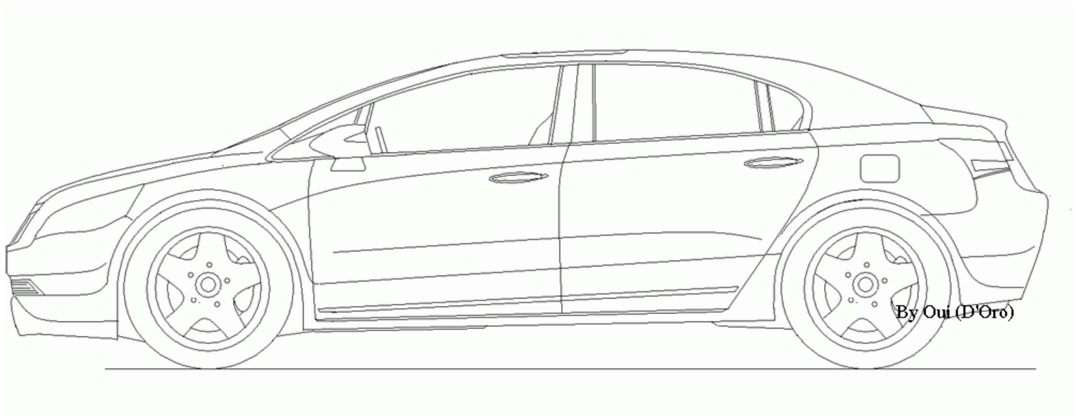
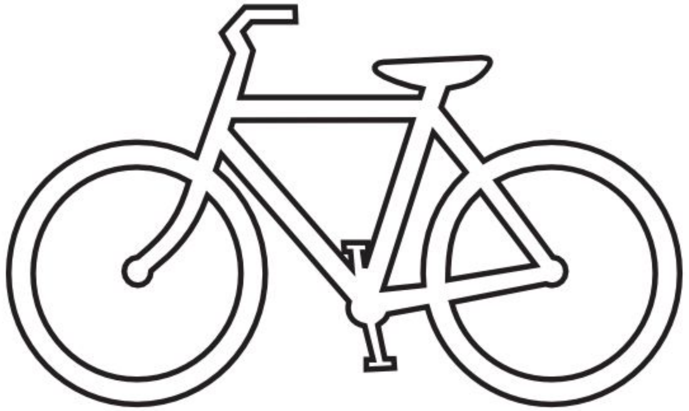
Chapter 13 Abstract Classes & Interfaces

What is an Abstract Class?



Incomplete
Share common design

What is an Abstract Class?



Incomplete
Share common design

Syntax : Abstract class, Abstract method

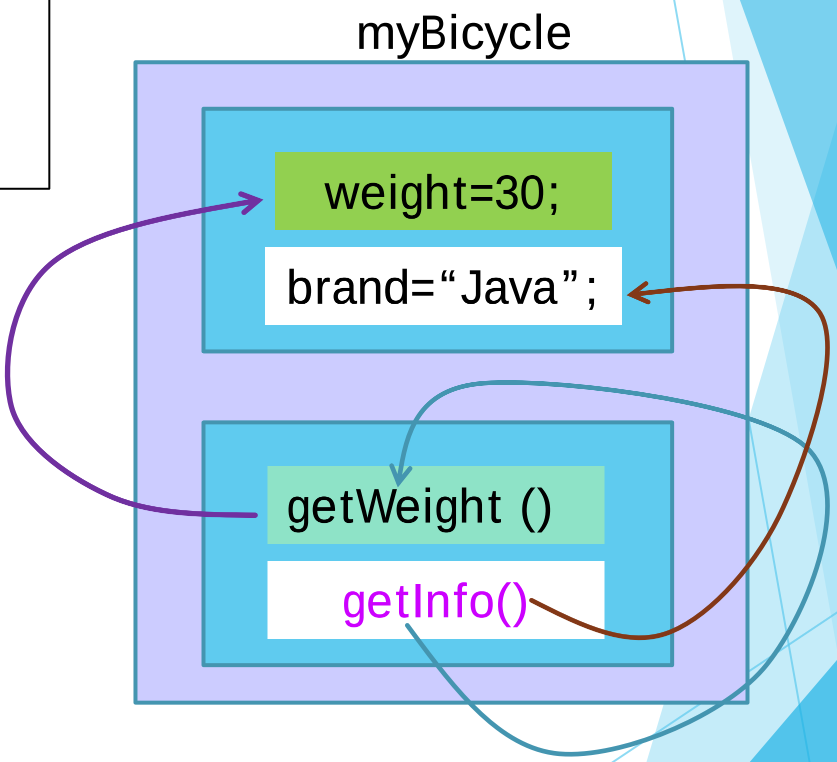
```
abstract <modifier> className {
     instance/static variables
     static/instance methods
     constructors
     abstract <modifier> void/type methodName();
}
```

How to declare an Abstract Class?

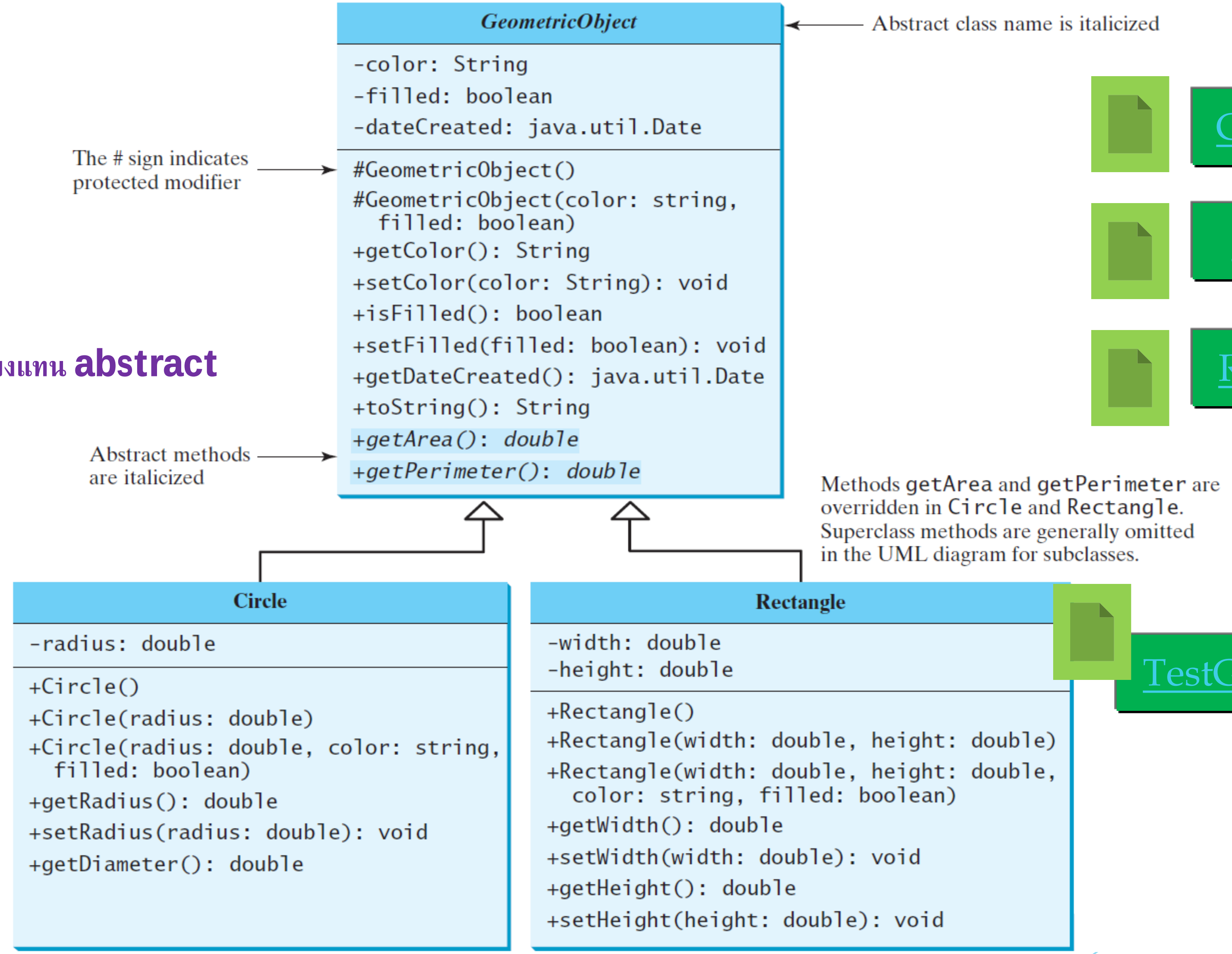
```
abstract public class Cycle {
   private int weight =30;
   public int getWeight(){ return weight;}
   abstract public void getInfo();

}
```

```
public class Bicycle extends Cycle{
   String brand = “Java”;
   public Bicycle(int w){
   public void getInfo(){
     sout(brand+”,”+getWeight());
   }
}
```

```
Bicycle myBicycle = new Bicycle();
myCycle.getInfo();
```



Abstract Classes and Abstract Methods



GeometricObject
-color: String
-filled: boolean
-dateCreated: java.util.Date
#GeometricObject()
#GeometricObject(color: string, filled: boolean)
+getColor(): String
+setColor(color: String): void
+isFilled(): boolean
+setFilled(filled: boolean): void
+getDateCreated(): java.util.Date
+toString(): String
+getArea(): double
+getPerimeter(): double
Abstract class name is italicized
The # sign indicates protected modifier
Abstract methods are italicized
Circle
-radius: double
+Circle()
+Circle(radius: double)
+Circle(radius: double, color: string, filled: boolean)
+getRadius(): double
+setRadius(radius: double): void
+getDiameter(): double
Rectangle
-width: double
-height: double
+Rectangle()
+Rectangle(width: double, height: double)
+Rectangle(width: double, height: double, color: string, filled: boolean)
+getWidth(): double
+setWidth(width: double): void
+getHeight(): double
+setHeight(height: double): void
Methods getArea and getPerimeter are overridden in Circle and Rectangle. Superclass methods are generally omitted in the UML diagram for subclasses.

ใช้สัญญลักษณ์ ตัวเอียงแทน **abstract**

GeometricObject

Circle

Rectangle

TestGeometricObject

Run

หลักการของ Abstract class

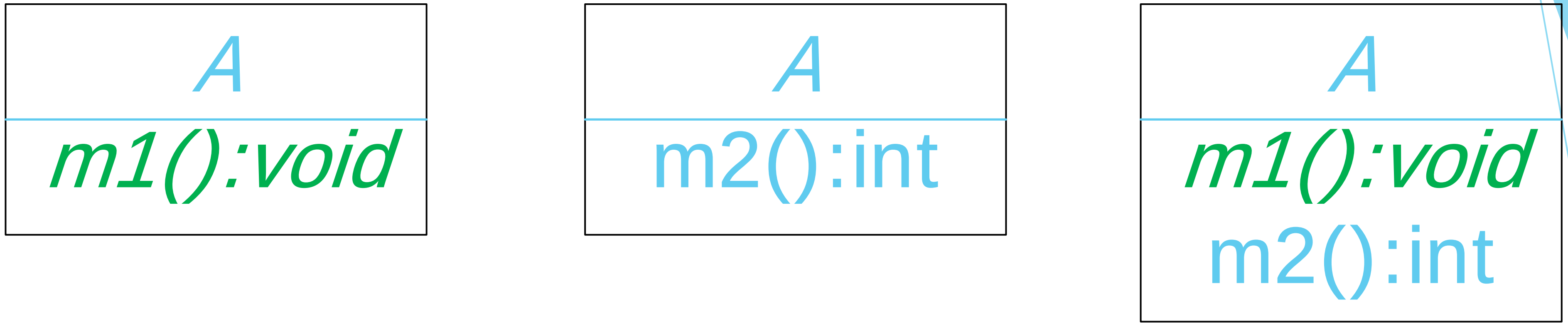
- ห้ามสร้าง object จาก abstract class : ใช้คำสั่ง new ไม่ได้

```
Cycle obj = new Cycle(); // ไม่ได้ๆๆๆ
```

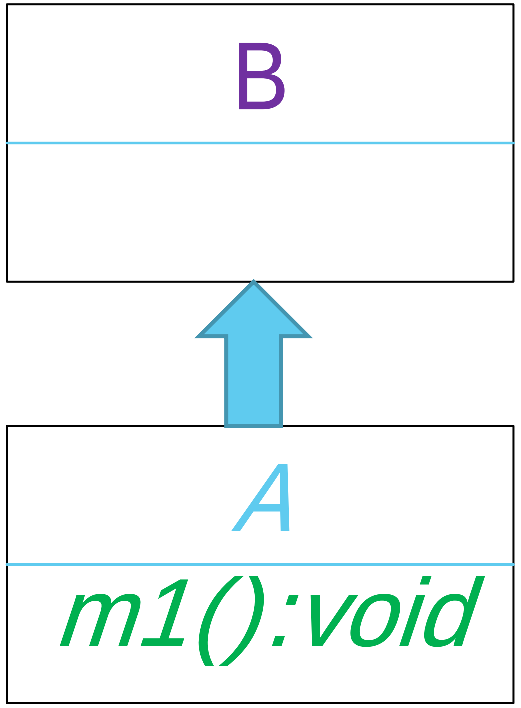
- Abstract class ถือว่ายังไม่สมบูรณ์ ออกแบบมาเพื่อ แชร์ common design
- ทำหน้าที่เป็น superclass ดังนั้น Constructors ของ abstract class ยังคงถูกเรียกโดย subclass ตามปกติ

หลักการของ Abstract class

- เป็น Supertype ได้ตามปกติ
  - *Cycle* obj = {new Bicycle(), new Bicycle()};
- คงมี dynamic binding และ polymorphism ตามปกติ
- Abstract class อาจจะไม่มี abstract methods ก็ได้
- Superclass ของ abstract class อาจเป็น concrete class ได้ (Object)
- ถ้า subclass ใดๆ สืบทอดต่อจาก abstract class อาจจะเป็น abstract class ต่อไปหรืออาจเป็น concrete class ได้แล้วแต่กรณี

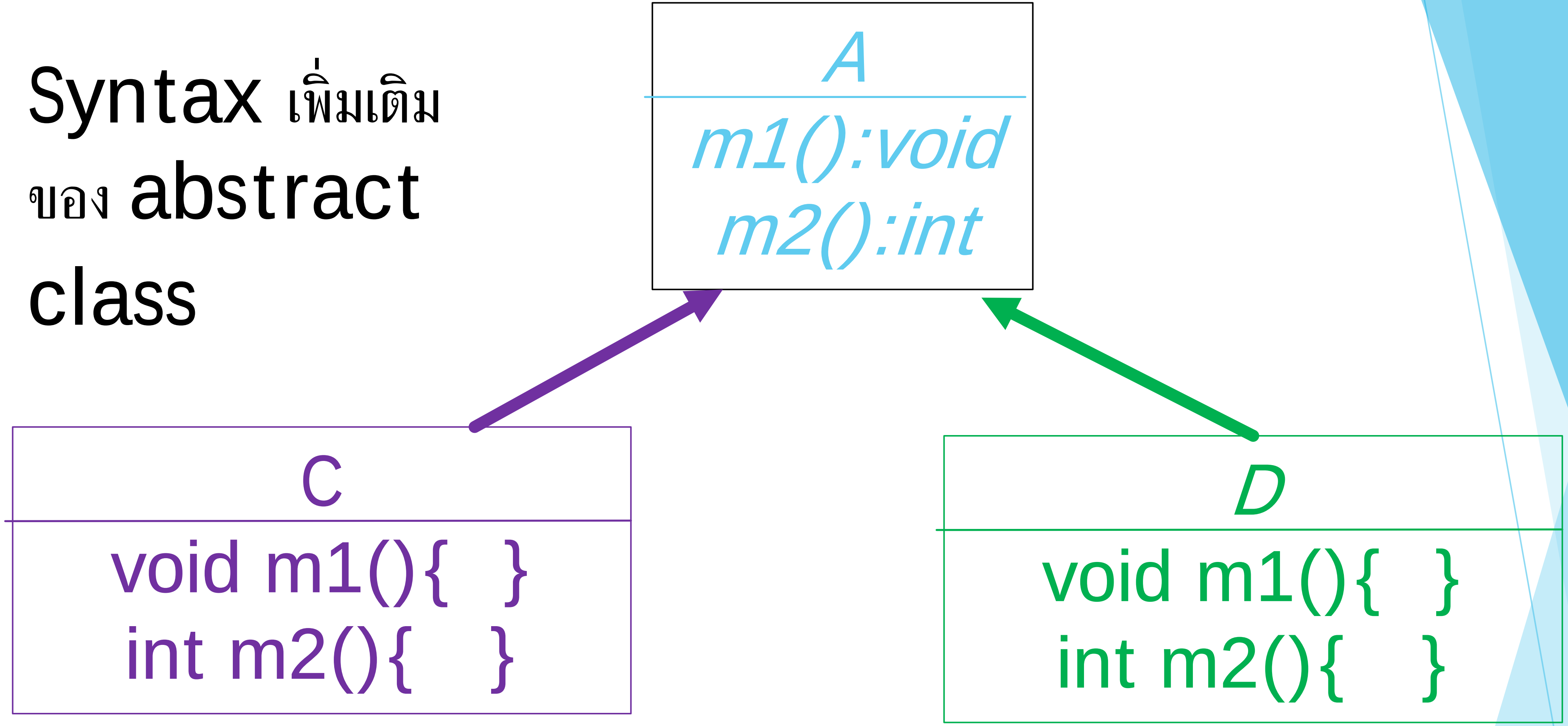


Syntax เพิ่มเติมของ abstract class



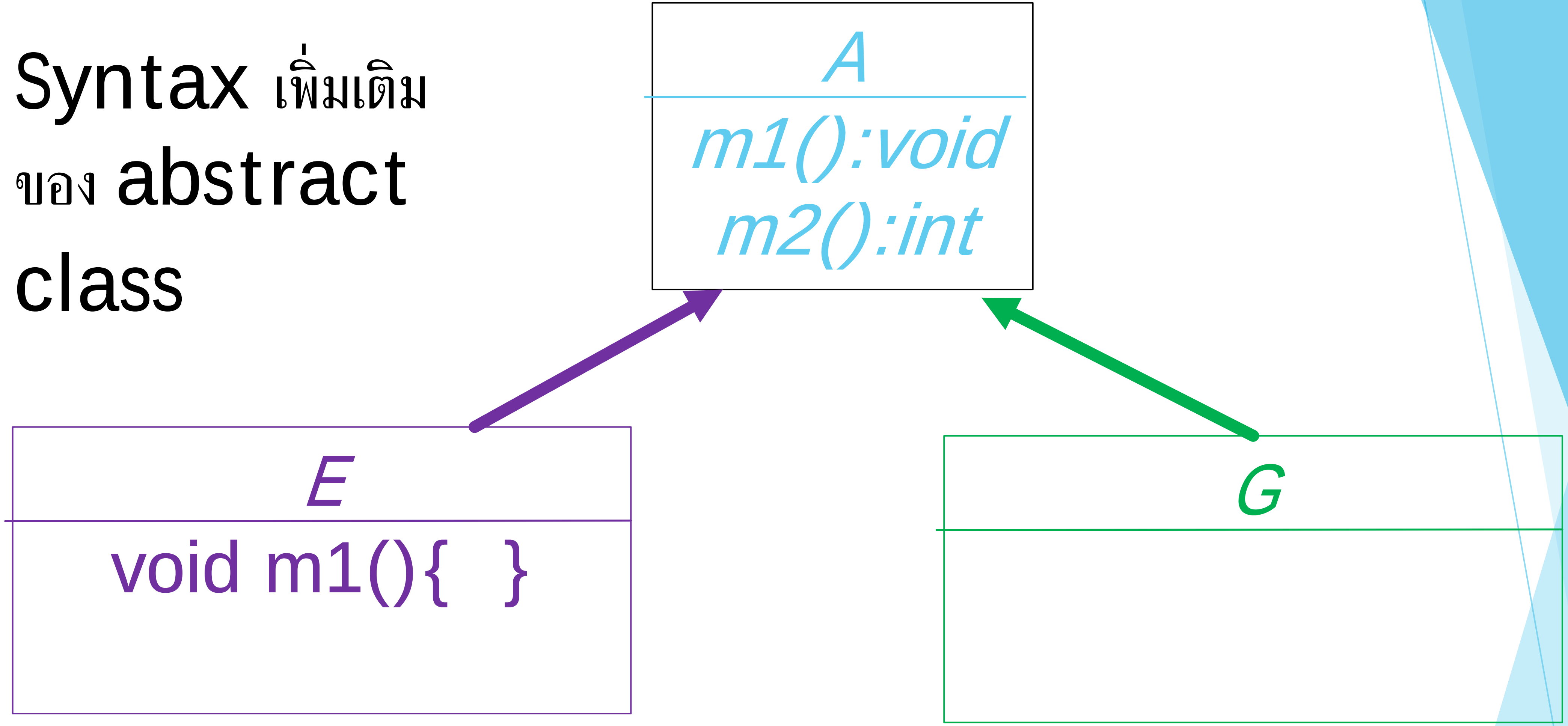
Superclass ของ abstract class อาจเป็น concrete class ได้

Syntax เพิ่มเติมของ abstract class



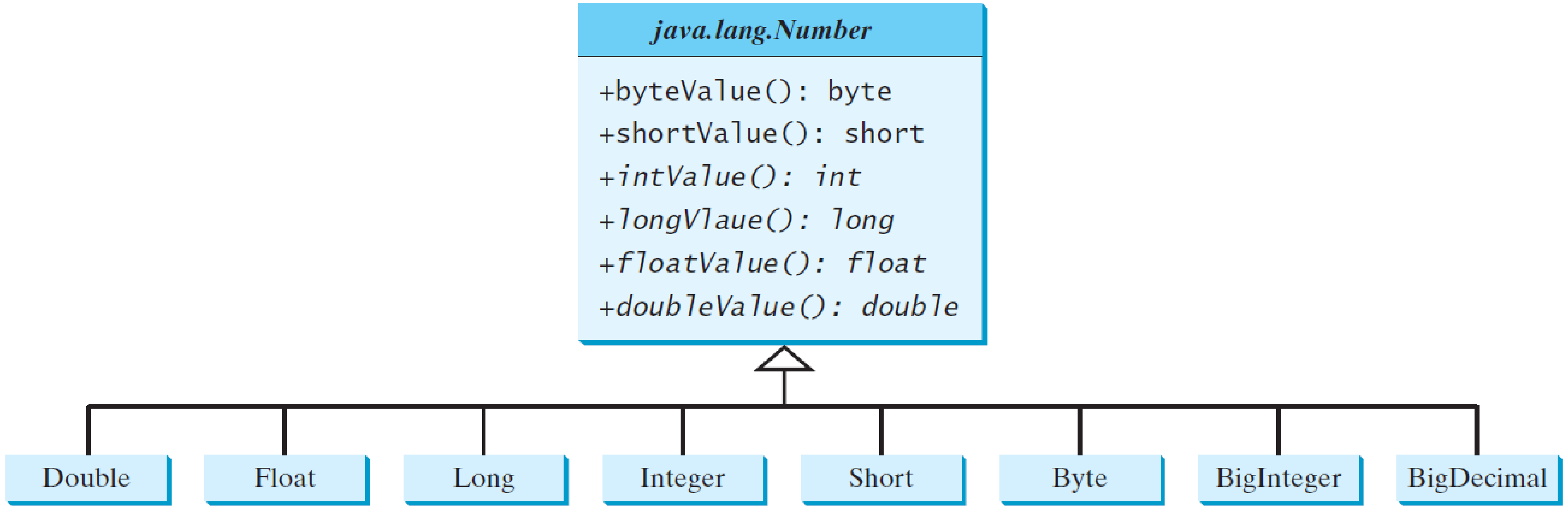
Subclass ได้ Implement ส่วน **Abstract methods ทั้งหมด**
Subclass อาจจะเป็น concrete class หรือ
Abstract class ก็ได้

Syntax เพิ่มเติมของ abstract class



Subclass ได้ Implement **Abstract methods** บางส่วน **หรือไม่ implement เลย**
Subclass ต้องเป็น Abstract class เท่านั้น

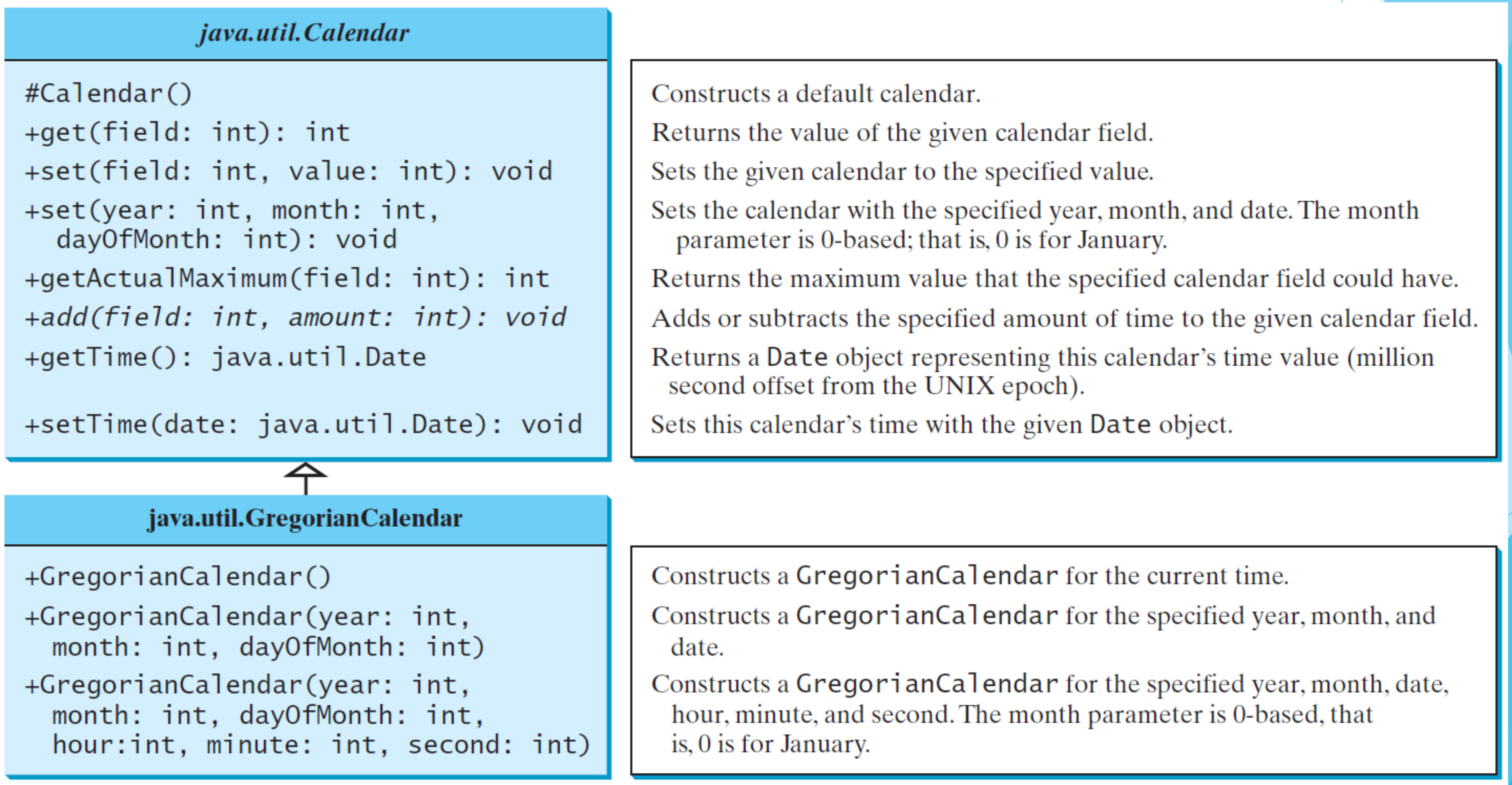
Case Study: the Abstract Number Class



LargestNumbers

Run

The Abstract Calendar Class and Its GregorianCalendar subclass



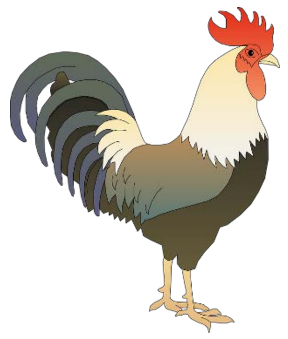
java.util.Calendar

`#Calendar()`	Constructs a default calendar.
`+get(field: int): int`	Returns the value of the given calendar field.
`+set(field: int, value: int): void`	Sets the given calendar to the specified value.
`+set(year: int, month: int, dayOfMonth: int): void`	Sets the calendar with the specified year, month, and date. The month parameter is 0-based; that is, 0 is for January.
`+getActualMaximum(field: int): int`	Returns the maximum value that the specified calendar field could have.
`+add(field: int, amount: int): void`	Adds or subtracts the specified amount of time to the given calendar field.
`+getTime(): java.util.Date`	Returns a `Date` object representing this calendar's time value (million second offset from the UNIX epoch).
`+setTime(date: java.util.Date): void`	Sets this calendar's time with the given `Date` object.

java.util.GregorianCalendar

`+GregorianCalendar()`	Constructs a `GregorianCalendar` for the current time.
`+GregorianCalendar(year: int, month: int, dayOfMonth: int)`	Constructs a `GregorianCalendar` for the specified year, month, and date.
`+GregorianCalendar(year: int, month: int, dayOfMonth: int, hour:int, minute: int, second: int)`	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.

What is Interface?



Loop:
    howtoEat() → Dynamic binding, Polymorphism

พวงมาลัย เบรค เกียร์

Loop:

howtoControl() → Dynamic binding, Polymorphism

What is Interface?

- คล้ายคลาส ภายใน Interface ประกอบ
  - Abstract methods
    - `abstract public void/type methodName();`
  - ค่าคงที่ในรูปแบบ
    - `public static final datatype variableName;`
- ไฟล์นามสกุล .java คอมไพล์แล้วได้ .class

Syntax : Interface

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

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

Syntax : การใช้งาน Interface ด้วยคำสั่ง **implements**

```
public class Orange implements Edible{
     public String howToEat(){
          return “orange juice”;
     }
}
```

ทำไมต้องมี Interface

- เพื่อกำหนดพฤติกรรมที่เป็น common ใช้ร่วมกัน
- เป็น data type ของตัวแปร object ที่เกิดจากคลาสที่มา implement ได้
  - `Edible obj = new Orange();`
- ห้ามใช้คำสั่ง new ในการเรียกใช้ Interface
  - `Edible obj = new Edible(); // ไม่ได้ๆๆๆ`

หลักการของ Interface

- Multiple inheritance
  - หนึ่งคลาส สามารถ implements ได้มากกว่า 1 interfaces

```
public class Orange extends Fruit implements
                                     Edible, Comparable{
        public String howToEat(){
             return “orange juice”;
        }
        public int compareTo(Object o){     }
}
```

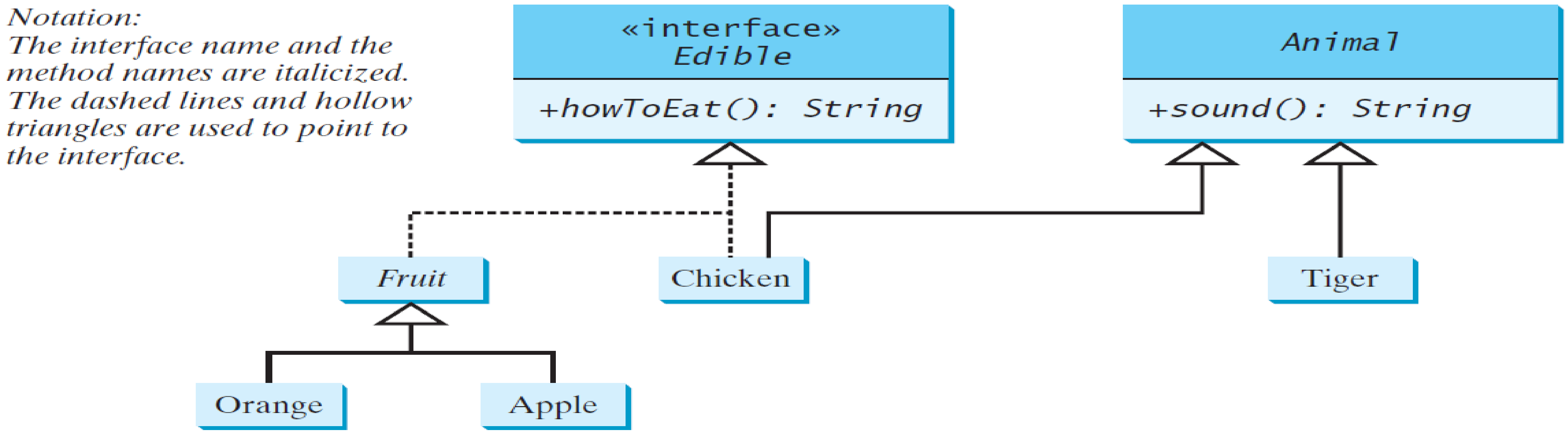
หลักการของ Interface :instanceof

```
public class Test{
    public static void main(String[] args){
        Orange obj = new Orange();
        if (obj instanceof Orange) {}
        if (obj instanceof Fruit) {}
        if (obj instanceof Object) {}
        if (obj instanceof Edible) {}
        if (obj instanceof Comparable) {}
    }
}
```

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

Edible TestEdible Run


Notation:
The interface name and the method names are italicized. The dashed lines and hollow triangles are used to point to the interface.
«interface»
Edible
+howToEat(): String
Animal
+sound(): String
Fruit
Chicken
Tiger
Orange
Apple

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

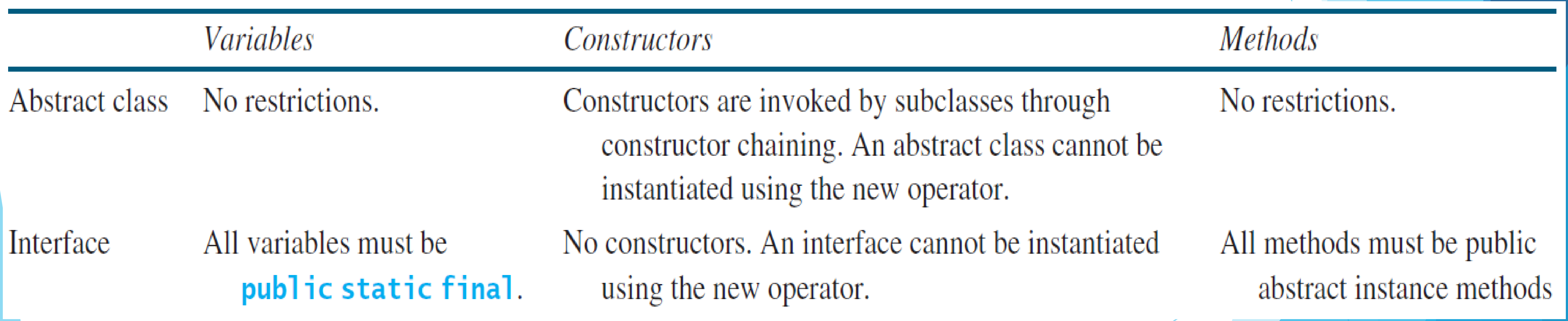
```
public interface T1 {
  int K = 1;

  void p();
}
```

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

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.

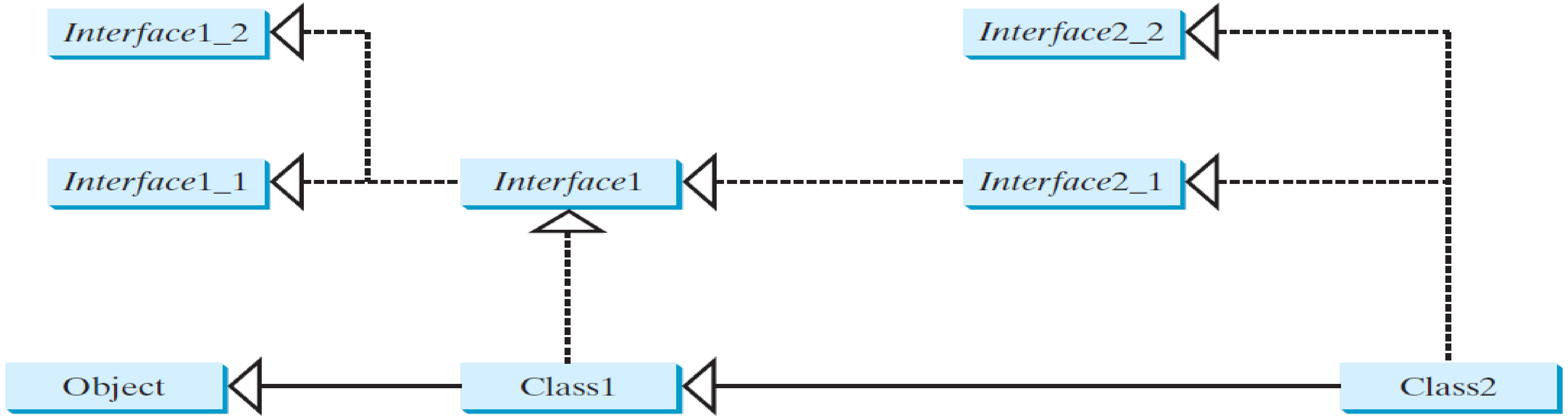


	Variables	*Constructors*	*Methods*
Abstract class	No restrictions.	Constructors are invoked by subclasses through constructor chaining. An abstract class cannot be instantiated using the new operator.	No restrictions.
Interface	All variables must be `public static final`.	No constructors. An interface cannot be instantiated using the new operator.	All methods must be public abstract instance methods

Interfaces vs. Abstract Classes, cont.

There is no single root for interfaces. If a class implements 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.

Interface1 extends *Interface1_1*,*Interface1_2* {   }



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.

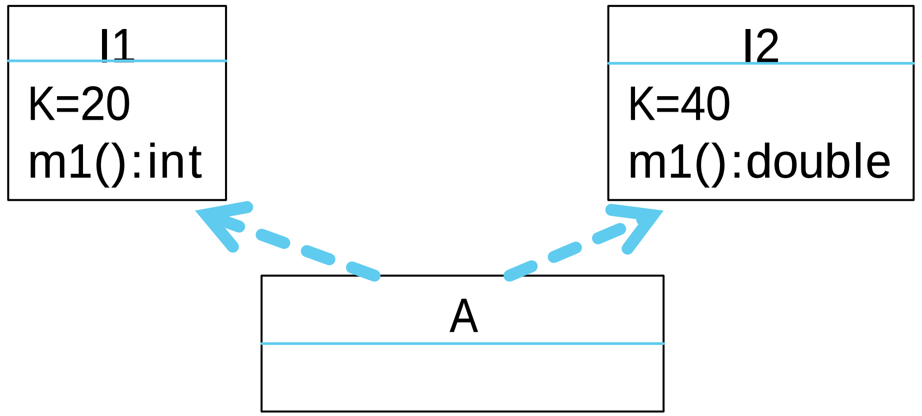
Caution: conflict interfaces

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).

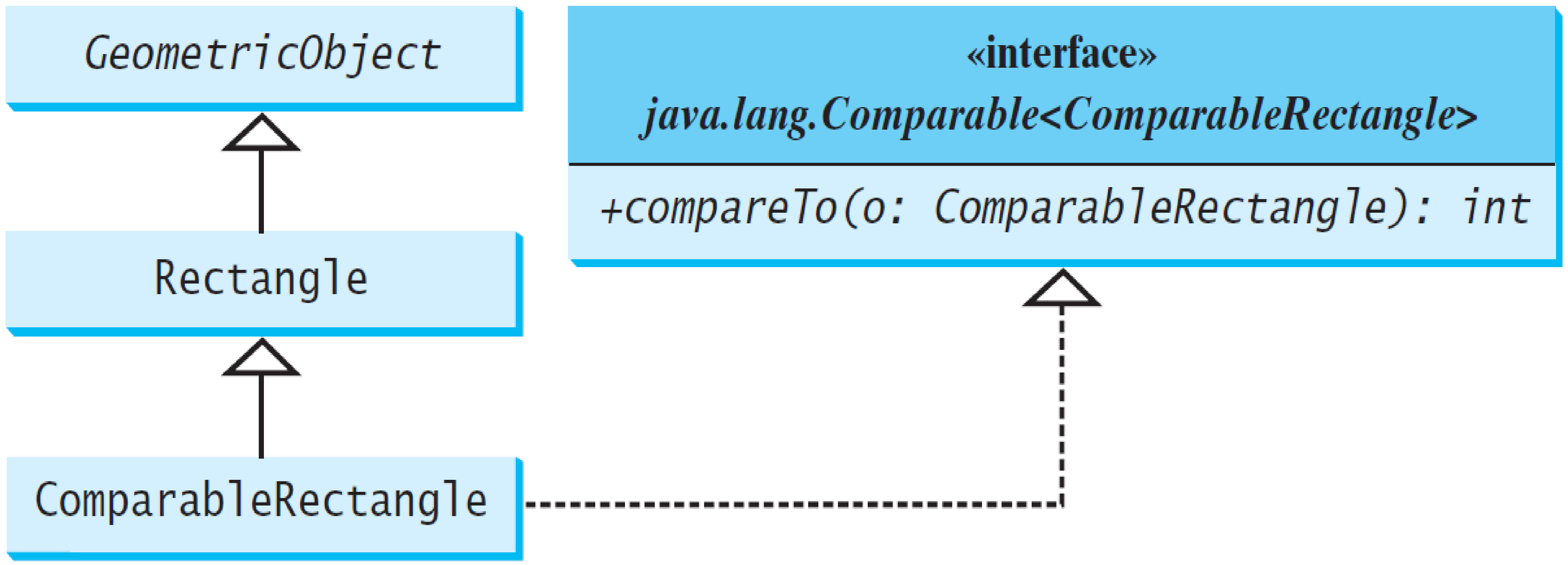
This type of errors will be detected by the compiler.



Example: The Comparable Interface

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

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

Defining Classes to Implement Comparable



GeometricObject
Rectangle
ComparableRectangle
«interface»
java.lang.Comparable<ComparableRectangle>
+compareTo(o: ComparableRectangle): int

ComparableRectangle

SortRectangles

Run

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 {
}
```

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 House implements Cloneable, Comparable<House>
{    private int id;
     private double area;
     private Date whenBuilt;


…
}
```

House

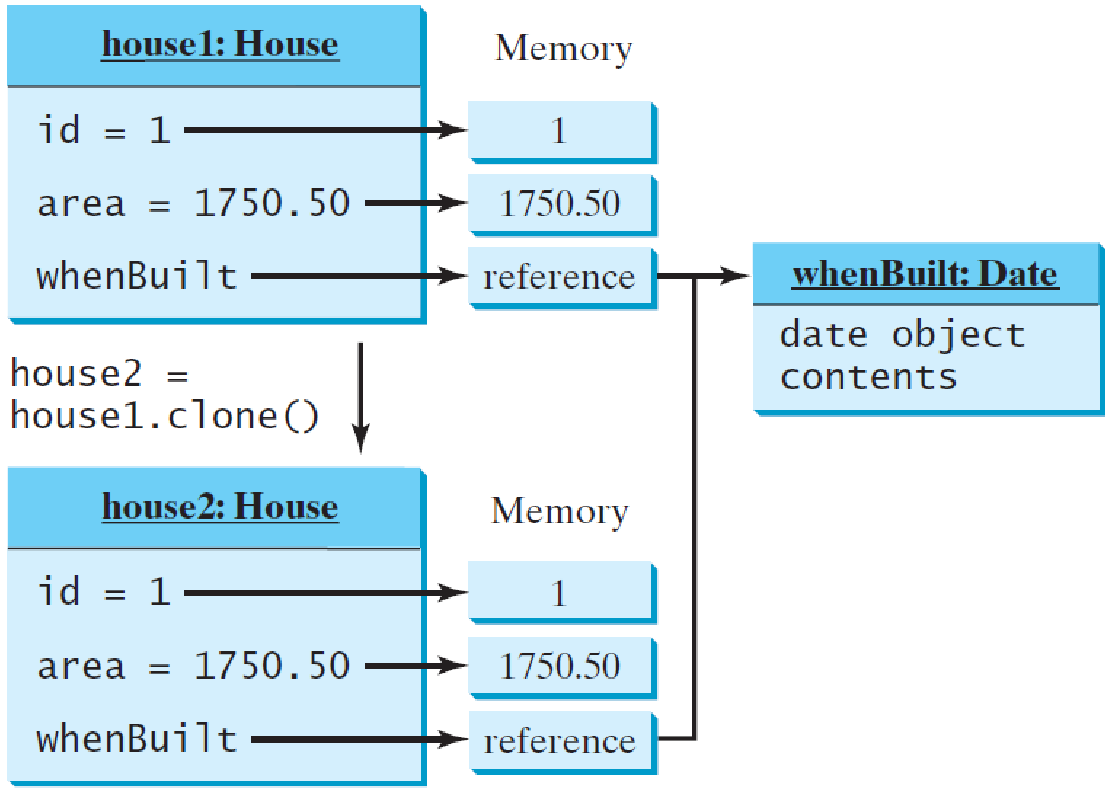
30

Shallow vs. Deep Copy

House house1 = new House(1, 1750.50);

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

Shallow Copy



(a)

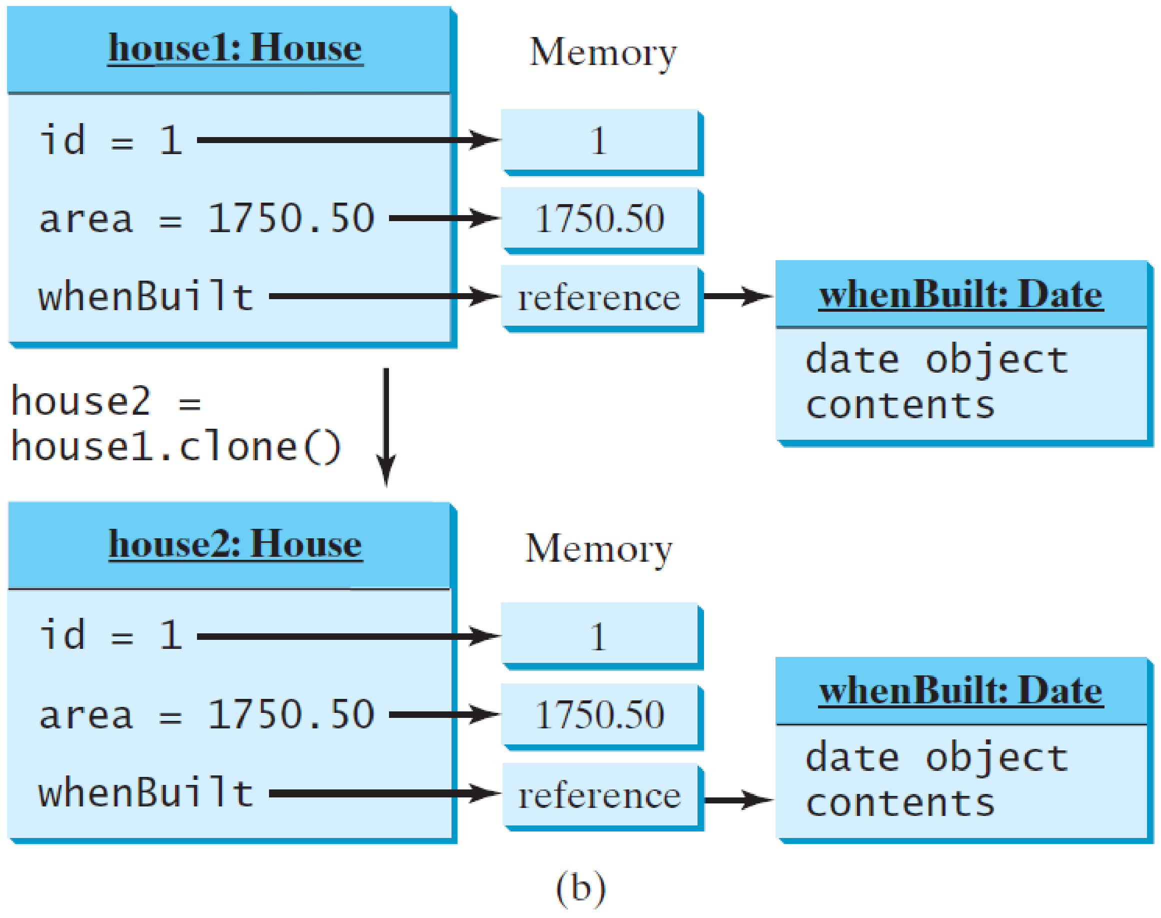
31

```
import java.util.Date;
public class House implements Cloneable, Comparable<House> {
  ….
   public Object clone() {

       try {
           return super.clone();

       } catch (CloneNotSupportedException ex) {
           return null;
       }
   }
  …
}
```

Shallow vs. Deep Copy

```
House house1 = new House(1, 1750.50);

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

Deep Copy



```
import java.util.Date;
public class House implements Cloneable, Comparable<House> {
  …
   public Object clone() {

       try {
              House houseClone = (House) super.clone();
              houseClone.whenBuilt = (Date) (whenBuilt.clone());
             return houseClone;
       } catch (CloneNotSupportedException ex) {
          return null;
       }
   }
…}
```