Abstract Classes and Interface

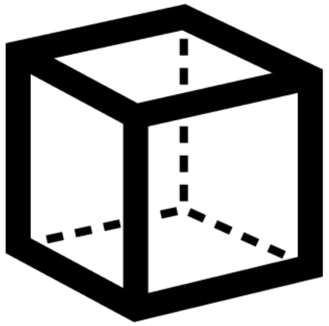
CSGE601021 Dasar-Dasar Pemrograman 2

Fakultas Ilmu Komputer Universitas Indonesia

References

- The slide is modified from
  Rahadianti, Laksmita. 2023. *Abstract Classes and Interface*. DDP2 B Genap 2022/2023.
- Liang, Introduction to Java Programming, 11th Edition, Ch. 13

Recall: Cube Object



UML Class Diagram

```
                Cube

           color: String
          length: double

               Cube()
Cube(color: String, length: double)
         toString(): String
         getVolume(): double
```

data fields

constructors

methods

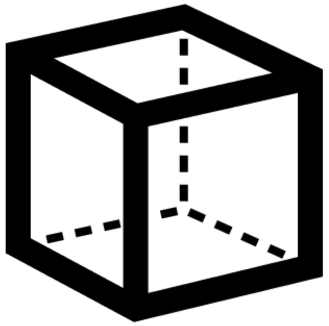
State: What are its properties?

- The state of an object consists of a set of data fields/variables.

Behavior: What does it do?

- The behavior of an object is defined by a set of methods.

Recall: Cube Object (2)



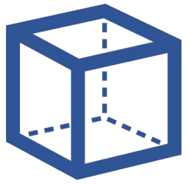
A class is a blueprint to create objects

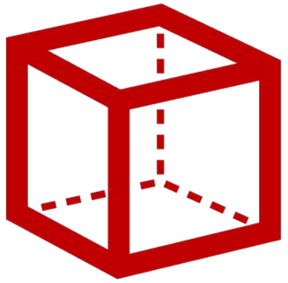
```
Cube

- color: String
- length: double
numOfCubes: int

Cube()
Cube(color: String, length: double)
+toString(): String
+getVolume(): double
+getNumOfCubes(): int
+equals(otherCube: Cube): Boolean
```

UML Objects

```
cube1: Cube

color = "Blue"
length = 1.0
```

```
cube2: Cube

color = "Red"
length = 2.0
```





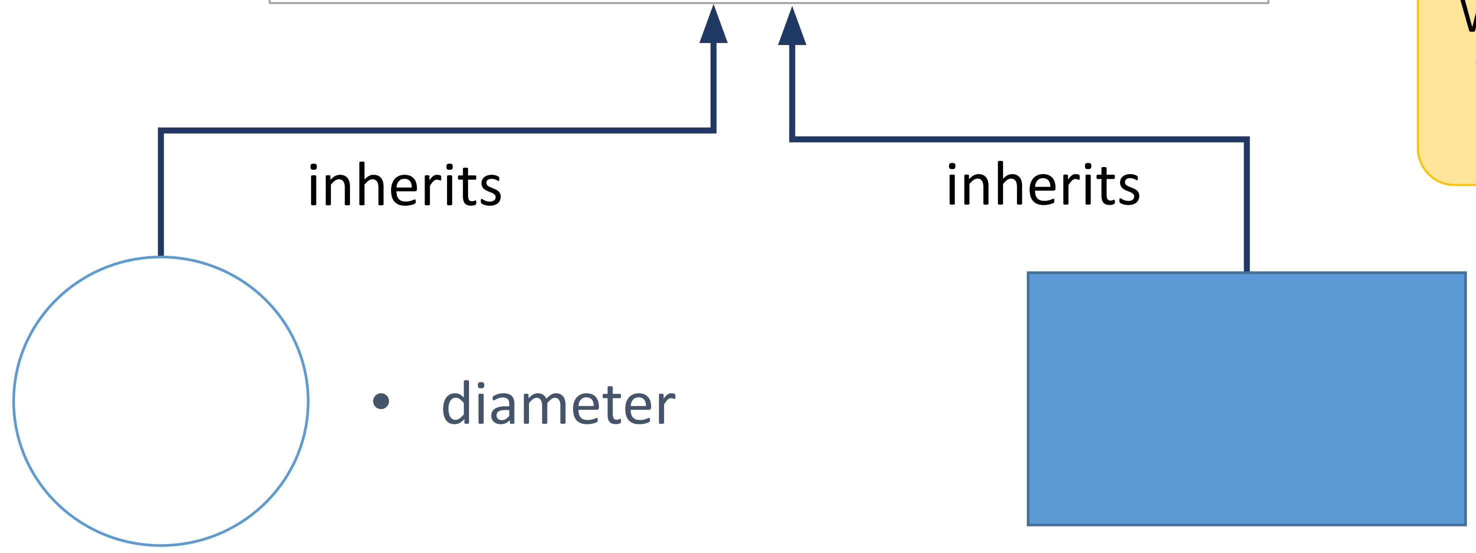
Inheritance

GeometricObject

- color
- filled

```
public class GeometricObject {
   private String color = "white";
   private boolean filled;
```

We design classes via inheritance to avoid redundancy



- diameter

- length
- height

Circle

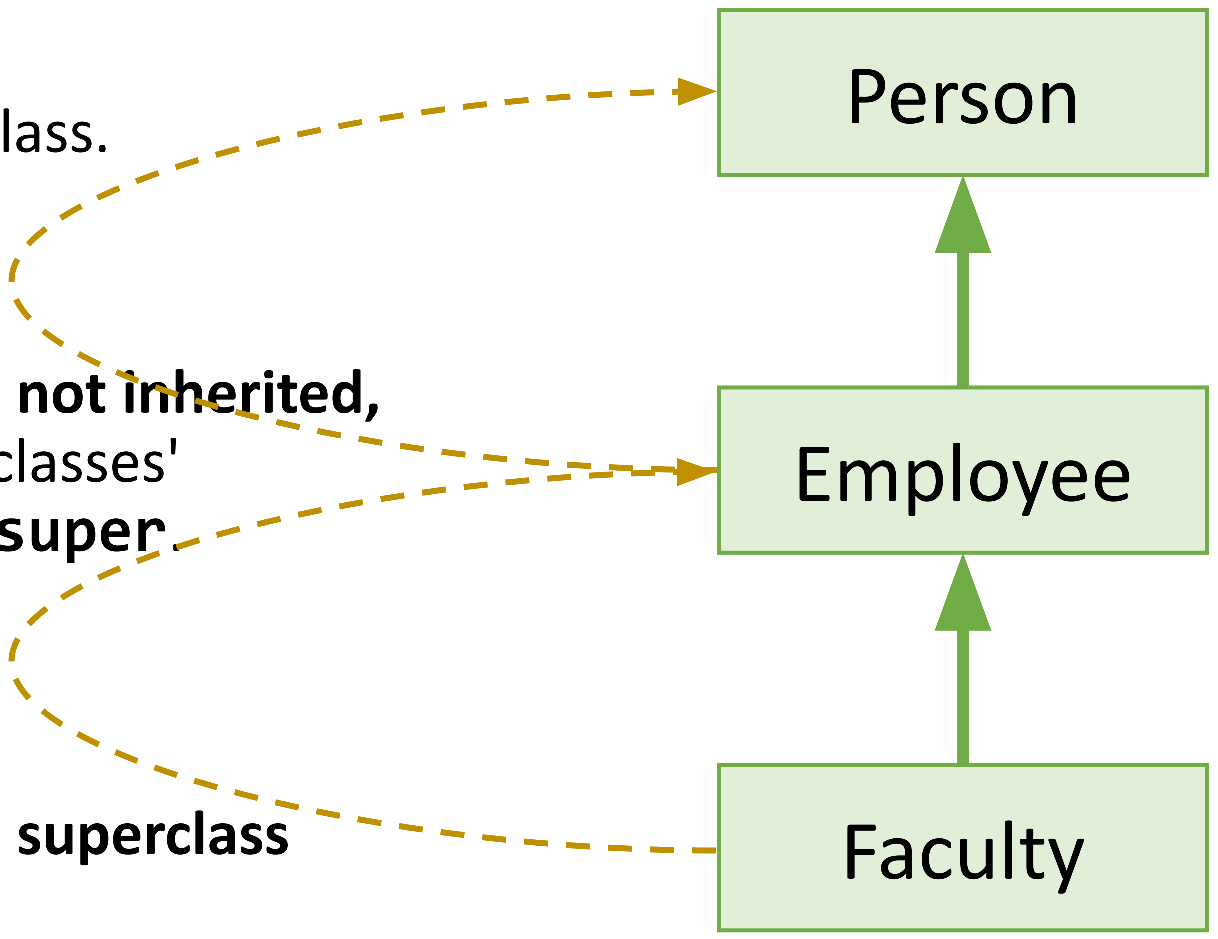
```
public class Circle extends GeometricObject {
  private double radius;
```

Rectangle

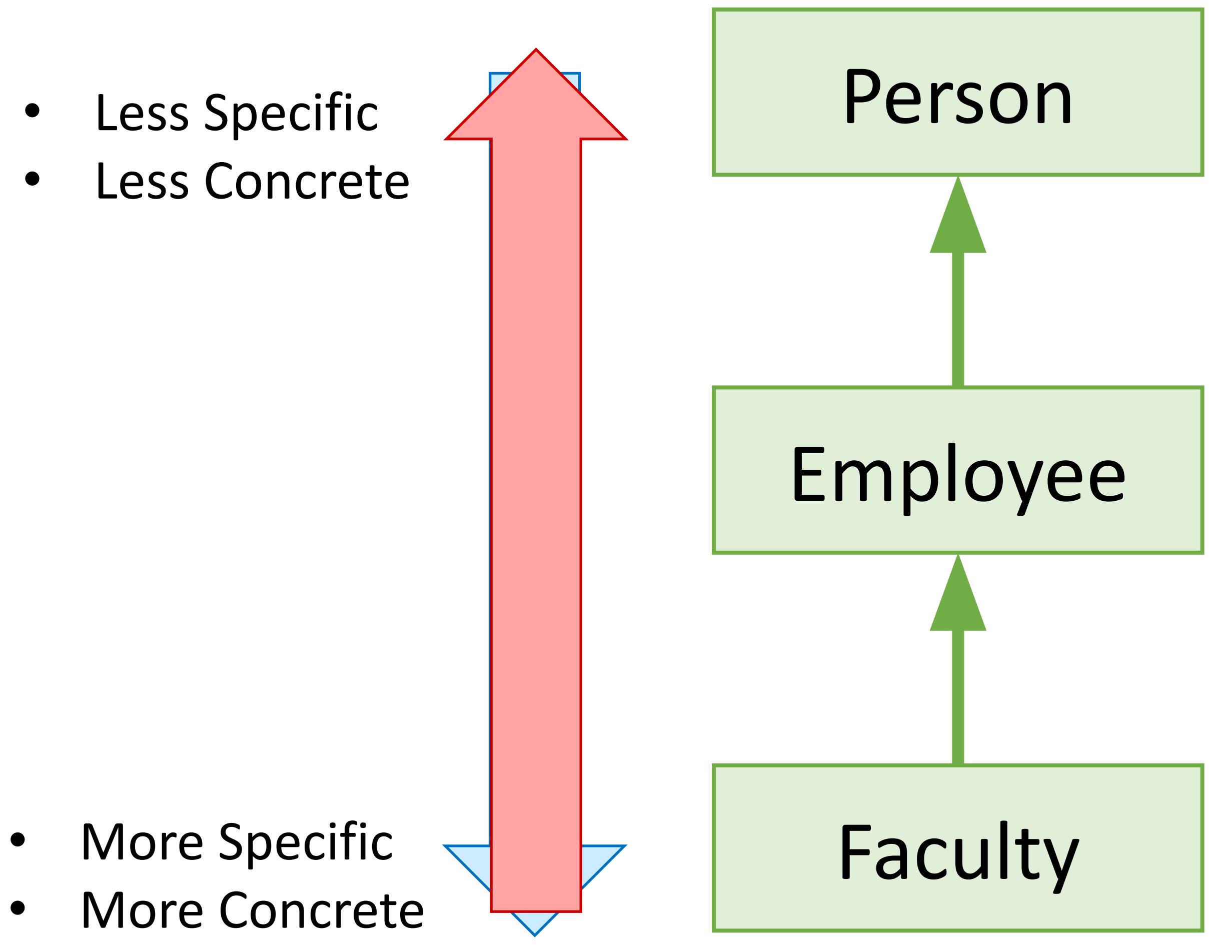
```
public class Rectangle extends GeometricObject {
  private double width;
  private double height;
```

Inheritance

- A subclass inherits from a superclass.
  - Superclass properties
  - Superclass methods
- The superclass’s constructors are **not inherited,** but can be invoked from the subclasses' constructors, using the keyword **super**.
- In the subclass, you can also:
  - Add new properties
  - Add new methods
  - **Override the methods of the superclass**



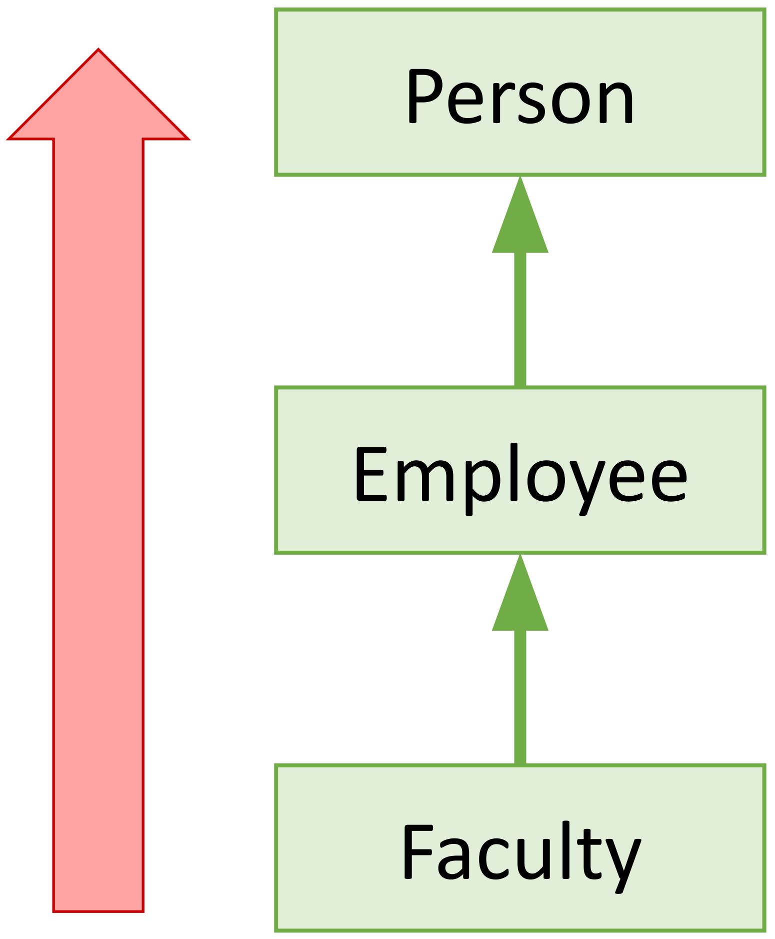
Superclasses and Subclasses



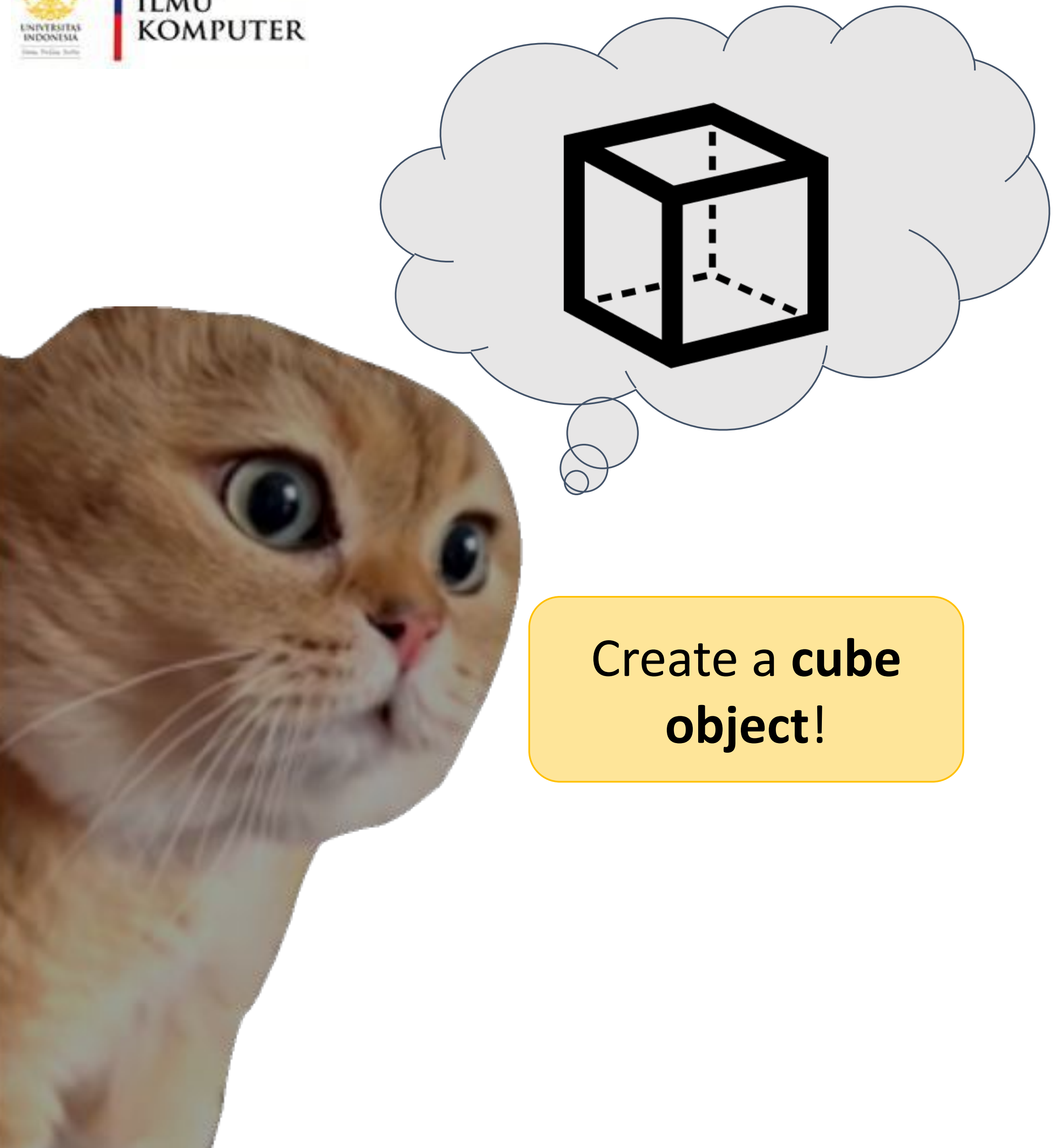
Superclasses and Subclasses

- Sometimes, a superclass **is so abstract,** it **cannot be used to create** any specific instances.

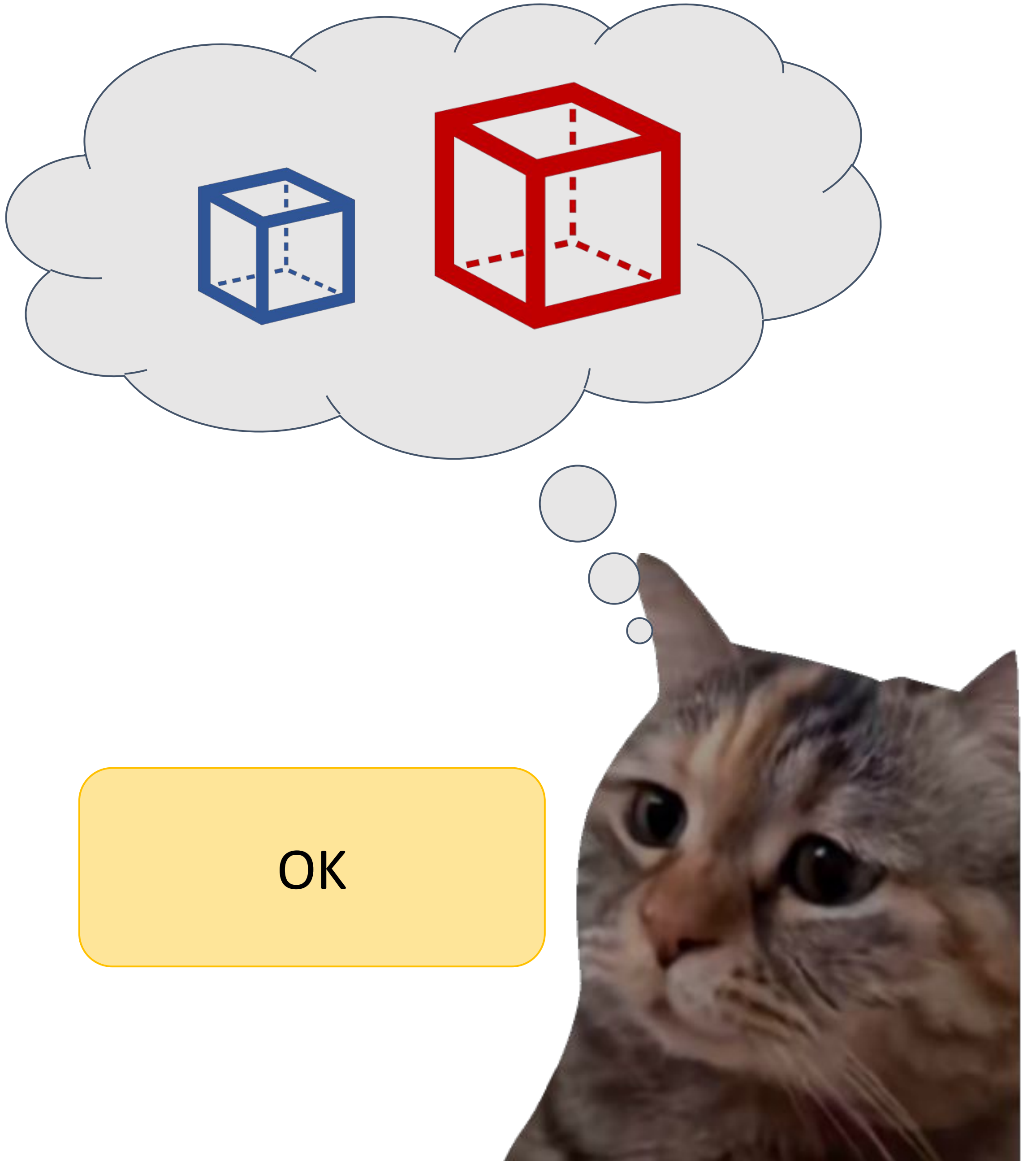
- Less Specific
- Less Concrete
- More **Abstract**



Abstract Classes

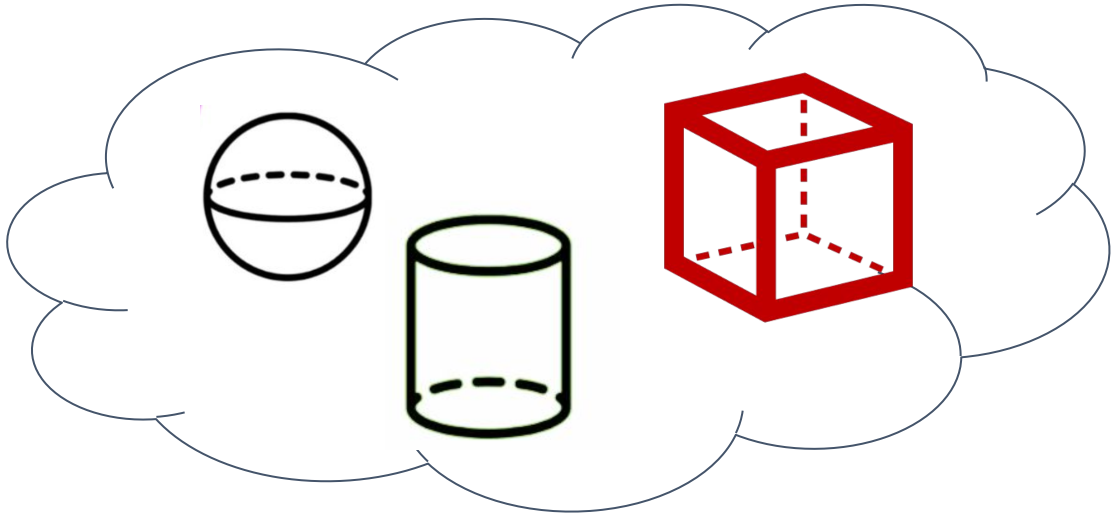


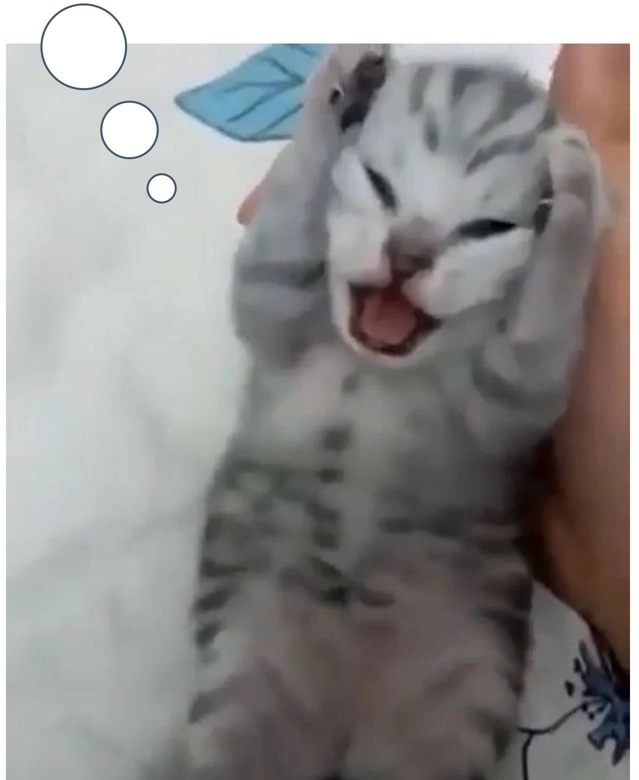
Create a **cube object**!



OK

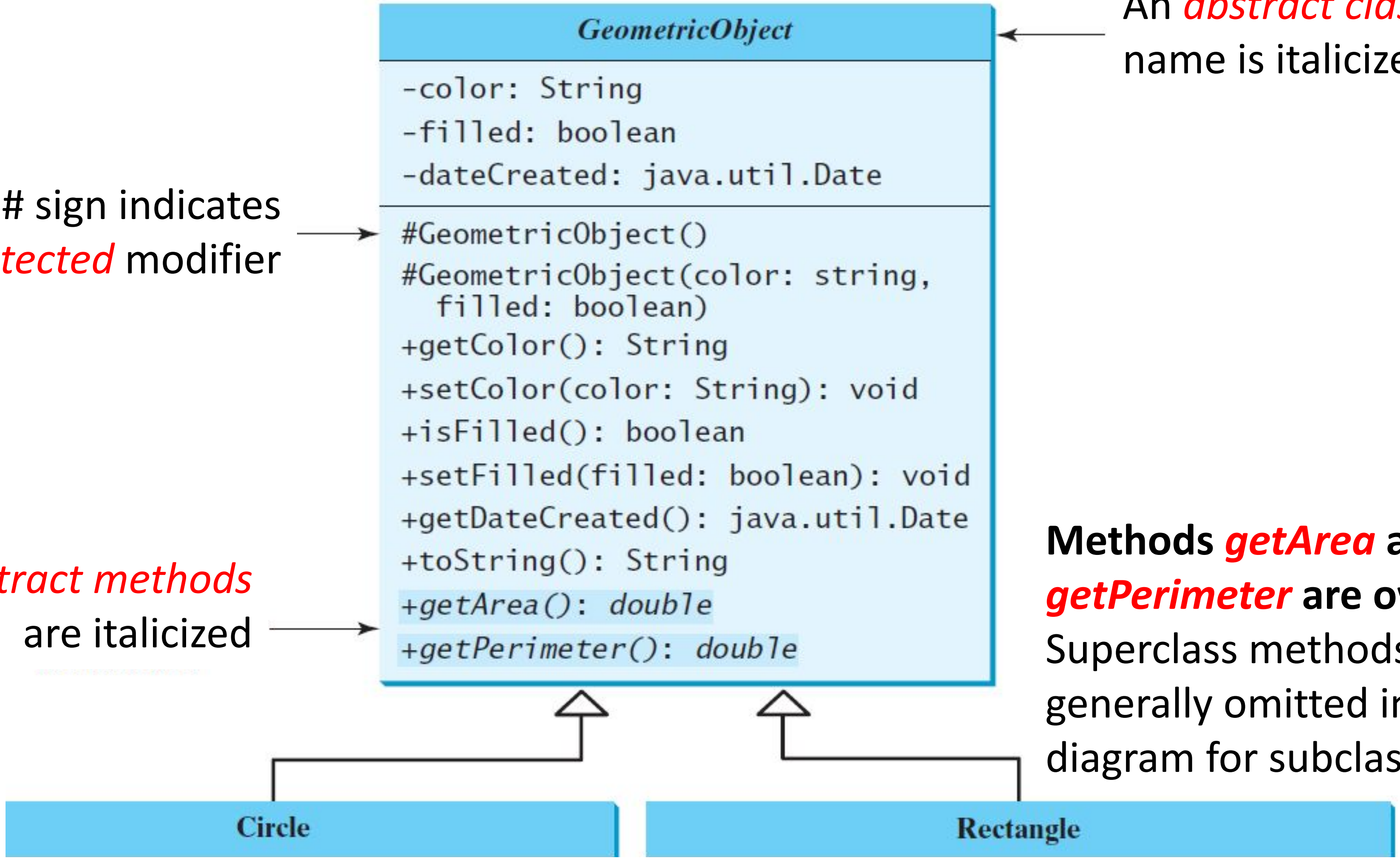
Create a **geometric object**!





Which one??? Too **abstract**!!

GeometricObject


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
Circle
Rectangle

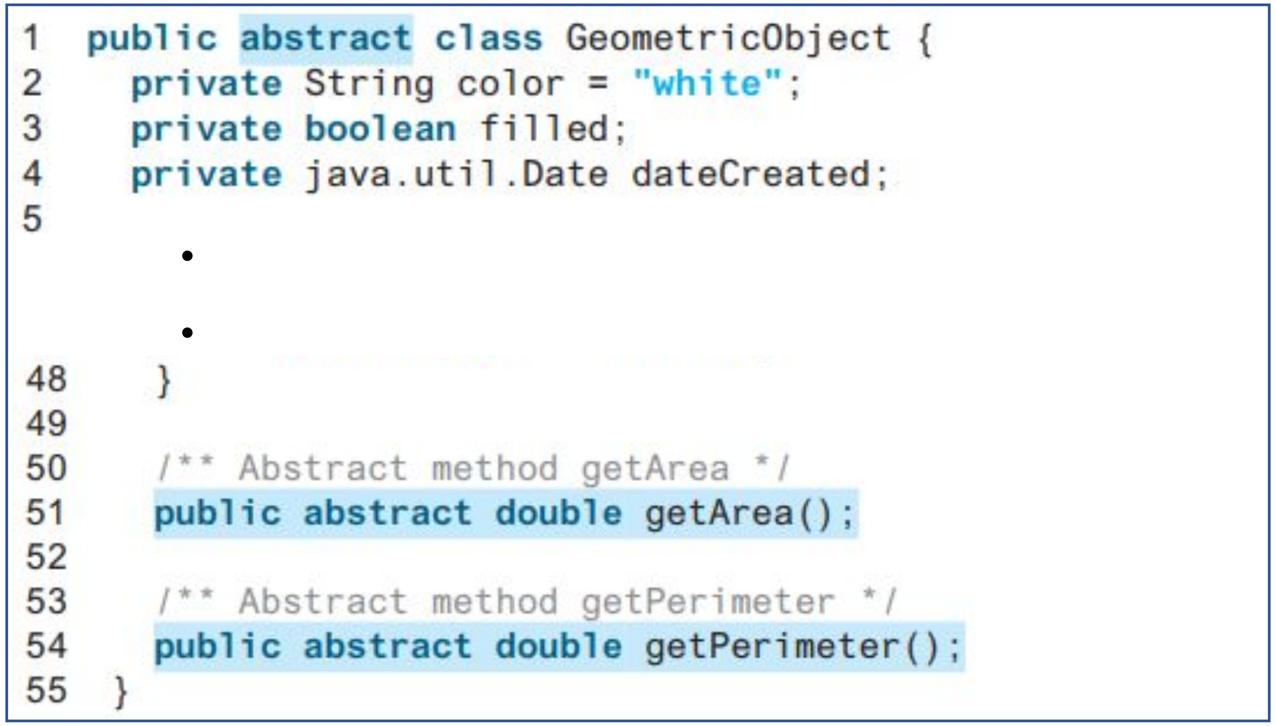
An *abstract class*'s name is italicized

The # sign indicates *protected* modifier

Abstract methods are italicized

Methods *getArea* and *getPerimeter* are overridden. Superclass methods are generally omitted in the UML diagram for subclasses.

Abstract Class: GeometricObject

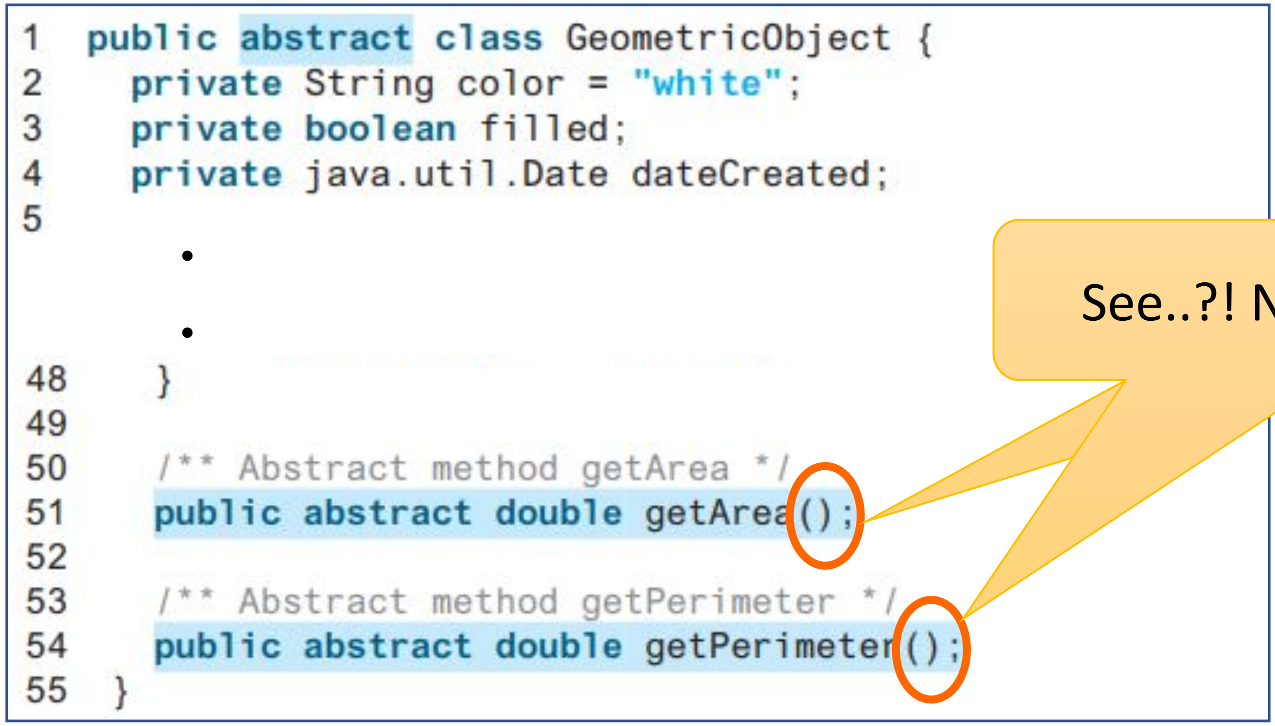


```java
public abstract class GeometricObject {
  private String color = "white";
  private boolean filled;
  private java.util.Date dateCreated;

       .

       .
     }

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

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

```java
public class Circle extends GeometricObject{
  // ...
}
```

```java
public class Rectangle extends GeometricObject{
  // ...
}
```

Abstract Methods



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

      .

      .
   }

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

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

See..?! No implementation

- Abstract method implementation depends on the specific type of object.
- A class extending an abstract class must implement all the abstract methods, unless the subclass is also abstract

Interesting Points about Abstract Classes

#1 Abstract methods in abstract classes

- vs Abstract classes without abstract methods

#2 Objects cannot be created from abstract classes

#3 Concrete methods can be overridden to be abstract

#4 Abstract class as type

#1 Abstract Method in Abstract Class

- An abstract method cannot be contained in a **non-abstract (or concrete) class**.
- An abstract class may have **abstract methods and/or non-abstract methods**.
- Which one is error-free?

```
class Shape {
  abstract double getPerimeter();
}
```

✗

```
abstract class Shape {
  abstract double getPerimeter();
  abstract double getArea();
}
```

✓

```
abstract class Shape {
  private String color;
  String getColor(){
    return color;
  }
}
```

✓

#1 Abstract Method in Abstract Class

- A class extending an abstract class must implement all the abstract methods, except the class is also abstract.

```
abstract class Shape {
  abstract double getPerimeter();
  abstract double getArea();
}
```

```
abstract class Rectangle extends Shape {
  abstract double getPerimeter();
  abstract double getArea();
}
```

✓

```
class Circle extends Shape {
  double diameter;
  double getDiameter() {...}
}
```

✗

```
class Triangle extends Shape {
  double getPerimeter() {...}
  double getArea() {...}
}
```

✓

#2 Objects cannot be created from abstract classes

- 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.
- This cannot be done.

```
abstract class Shape {
  private String color;

  Shape (String color) {
    this.color = color;
  }
}
```

```
class TestShape {
  Shape s = new Shape("green");
}
```

✗

#3 Concrete Methods can be Overridden to be Abstract

```
class Circle {
  private double radius;
  Circle(double radius) {
    this.radius = radius;
  }
  double getArea(){
    return Math.PI * radius * radius;
  }
}
```

```
abstract class Round extends Circle{
  double r1;
  double r2;
  Round(double r1, double r2){
    super(r1);
    this.r1 = r1;
    this.r2 = r2;
  }
  abstract double getArea();
}
```

- **A subclass can override a method from its superclass as abstract.** This is **rare**, but useful when the implementation of the method in the superclass becomes invalid in the subclass.
- A subclass can be abstract even if its superclass is concrete. (`Object` is concrete)

#4 Abstract Class as Type

- Which is error-free?

```
class TestShape {
  public static void main(String[] ar){
    Shape no = new Shape("red");
  }
}
```

✗

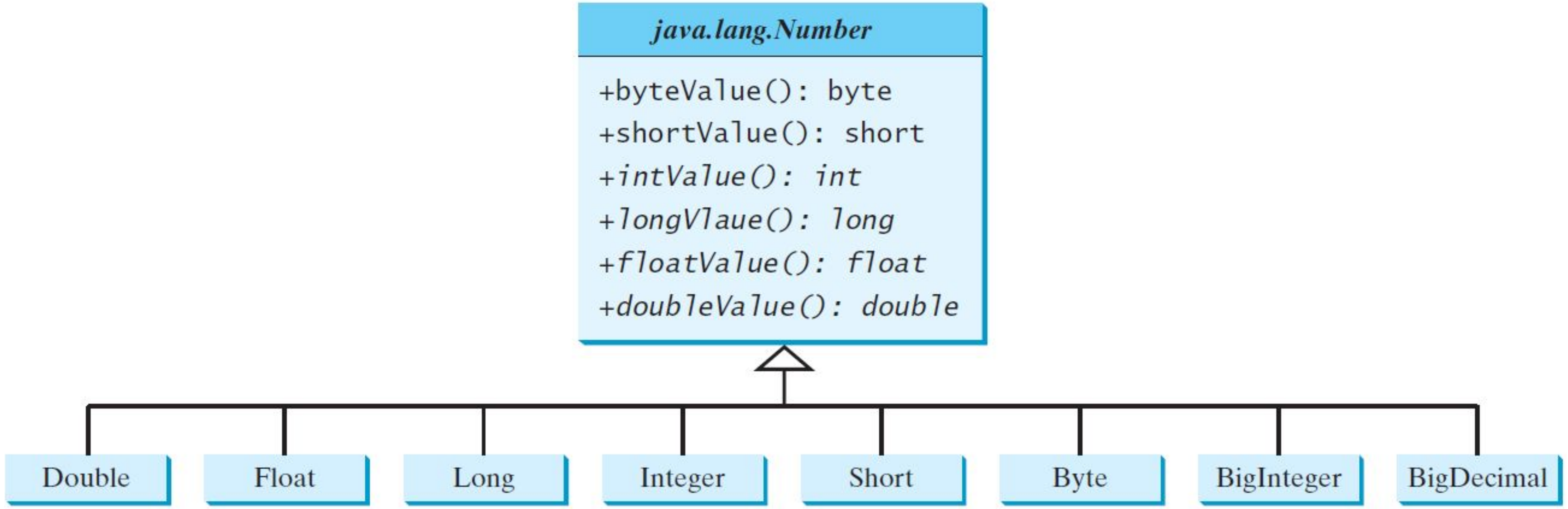
```
class TestShape {
  public static void main(String[] ar){
    Shape[] s = new Shape[3];
    s[0] = new Circle("red",2.3);
  }
}
```

✓

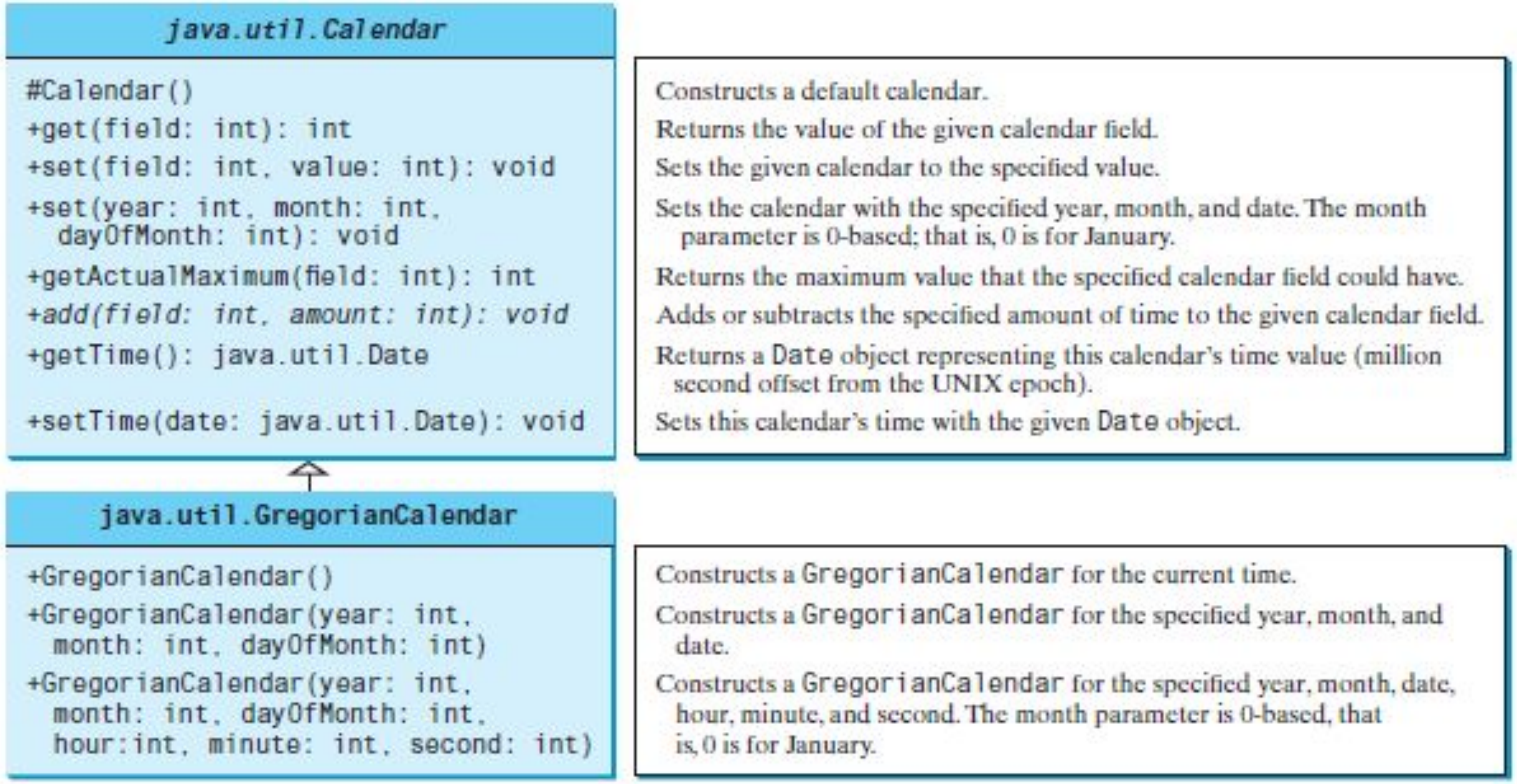
```
abstract class Shape {
  String color;
  Shape (String color) {
    this.color = color;
  }
}
class Circle extends Shape {
  private double radius;
  Circle(String color, double radius){
    super(color);
    this.radius = radius;
  }
}
```

- You cannot create an instance from an abstract class using the new operator, but an abstract class can be used as a data type.

Abstract class: Number



Abstract class: `Calendar`



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.

Interface

Interface

- An interface is a ***classlike*** construct that contains only constants and abstract methods, **similar to** an abstract class, but with the intent to specify common behavior for objects.
- You can specify that the objects are `Comparable`, `Edible`, `Cloneable` using appropriate interfaces
  - Car is **Rentable**
  - Mushroom is **Edible**

Interface is a Special Class

- Like an abstract class, you cannot create an instance from an interface using the new operator, but you can mostly use an interface more or less the same way you use an abstract class.
- Unlike an abstract class, **interface cannot have constructors**.
- To use an abstract class, a class `implements Interface`, <u>not</u> `extends Interface`

Defining an Interface

- To distinguish an interface from a class, Java uses the following syntax to define an interface:

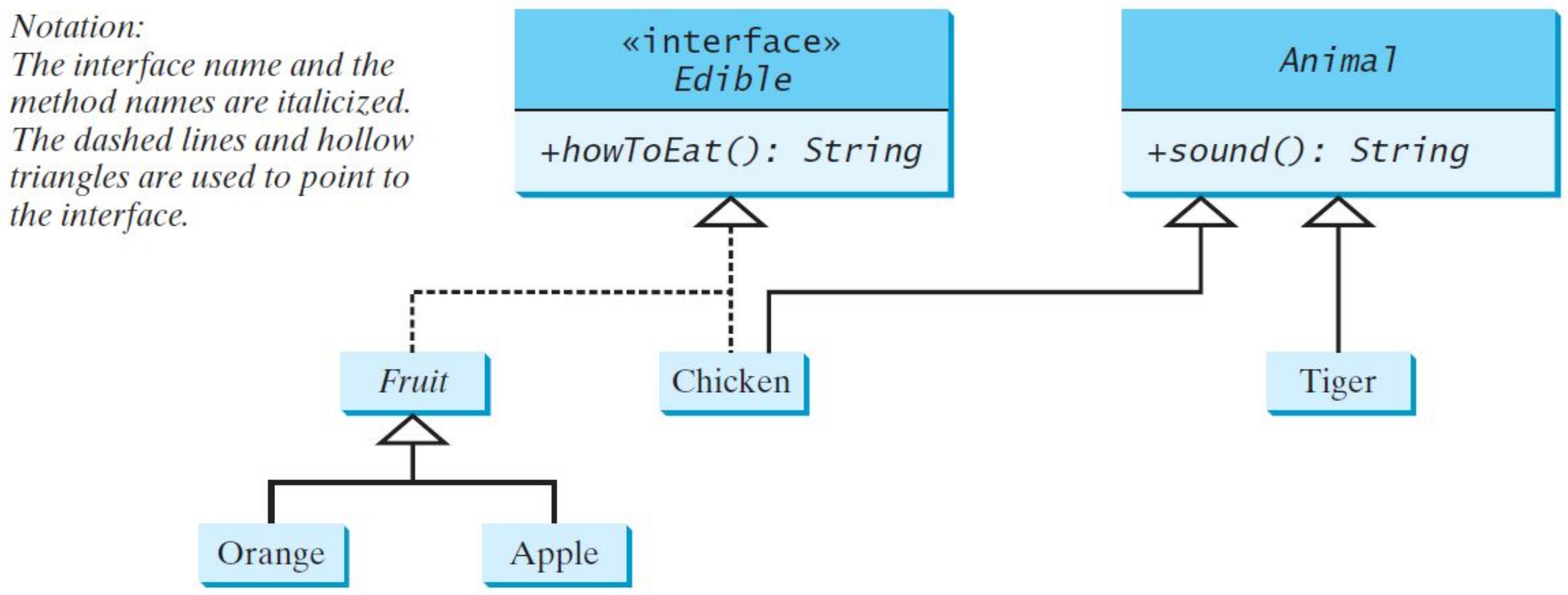
```
public interface InterfaceName {
  constant declarations;
  abstract method signatures;
}
```

- Example:

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

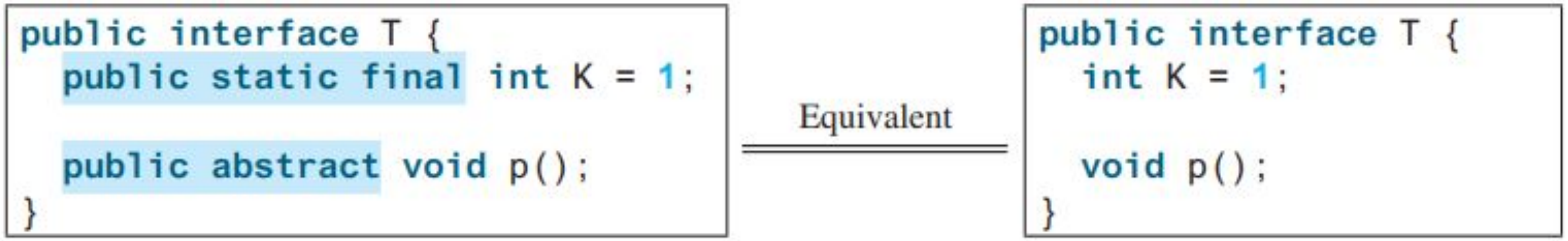
Example

- For example, the classes `Chicken` and `Fruit` implement the `Edible` interface


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`, all methods are `public abstract` in an interface.
- Modifiers can then be omitted, as shown below:



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

  public abstract void p();
}
```

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

  void p();
}
```

- Although the public modifier may be omitted for a method defined in the interface, the method must be defined public when it is implemented in a subclass.

Interfaces for Cellphone

```
// GPSEnabled.java
interface GPSEnabled {
    public void printLocation();
}

// RadioEnabled.java
interface RadioEnabled {
    public void startRadio();
    public void stopRadio();
}
```

```
public class Cellphone implements GPSEnabled,
RadioEnabled {
    public void printLocation() {
      System.out.println("Location");
    }
    public void startRadio() {
      System.out.println("Radio is ON!");
    }
    public void stopRadio() {
      System.out.println("Radio is OFF!");
    }
}
```

Object Cellphone (1)

```
// GPSEnabled.java
interface GPSEnabled {
    public void printLocation();
}

// RadioEnabled.java
interface RadioEnabled {
    public void startRadio();
    public void stopRadio();
}
```

```
public class Cellphone implements GPSEnabled,
RadioEnabled {
    public void printLocation() {
        System.out.println("Location");
    }
    public void startRadio() {
        System.out.println("Radio is ON!");
    }
    public void stopRadio() {
        System.out.println("Radio is OFF!");
    }
}
```

```
Cellphone ciaoMi = new Cellphone();
ciaoMi.printLocation();
ciaoMi.startRadio();
```

Object Cellphone (2)

```
// GPSEnabled.java
interface GPSEnabled {
    public void printLocation();
}

// RadioEnabled.java
interface RadioEnabled {
    public void startRadio();
    public void stopRadio();
}
```

```
public class Cellphone implements GPSEnabled,
RadioEnabled {
    public void printLocation() {
      System.out.println("Location");
    }
    public void startRadio() {
      System.out.println("Radio is ON!");
    }
    public void stopRadio() {
      System.out.println("Radio is OFF!");
    }
}
```

```
GPSEnabled samsu = new Cellphone();
samsu.printLocation();
samsu.startRadio();  ✗
```

Interface Methods

```
public interface RadioEnabled {
    public void startRadio();
    public void stopRadio();
    public void recordRadio(); // let’s add this


}
public class Cellphone implements RadioEnabled {
    public void startRadio() {
      // start radio
    }
    public void stopRadio() {
      // stop radio
    }
}
```

Any class that implements the interface **must** provide an implementation.

We must not forget!

`default` Interface Methods (Java 8+)

```
public interface RadioEnabled {
    public void startRadio();
    public void stopRadio();
    default public void recordRadio() {
        System.out.println("Recording radio..");
    }
}
public class Cellphone implements RadioEnabled {
    public void startRadio() {
      // start radio
    }
    public void stopRadio() {
      // stop radio
    }
}
```

If we provide a default implementation, it will be used unless overridden in the subclasses

`default` interface methods (Java 8+)

- A default method provides a default implementation for the method in the interface. A class that implements the interface may simply:
  - use the default implementation for the method, or
  - override the method with a new implementation.
- This feature enables you to add a new method to an existing interface with a default implementation without having to rewrite the code for the existing classes that implement this interface.

```java
public interface GPSEnabled {
    public void printLocation();
}

public interface RadioEnabled {
    public int printLocation();
    public void startRadio();
    public void stopRadio();
}

public class Cellphone implements GPSEnabled, RadioEnabled {
    public void printLocation() {
      // return location
    }

    // ...

}
```

```
public interface GPSEnabled {
    public void printLocation();
}

public interface RadioEnabled {
    public int printLocation();
    public void startRadio();
    public void stopRadio();
}

public class Cellphone implements GPSEnabled, RadioEnabled {
    public void printLocation() {
      // return location
    }

    // ...

}
```

What's wrong?
Name collision: Overlapped methods with different return types!

Some Common Interfaces

- Comparable
- Clonable
- Serializable

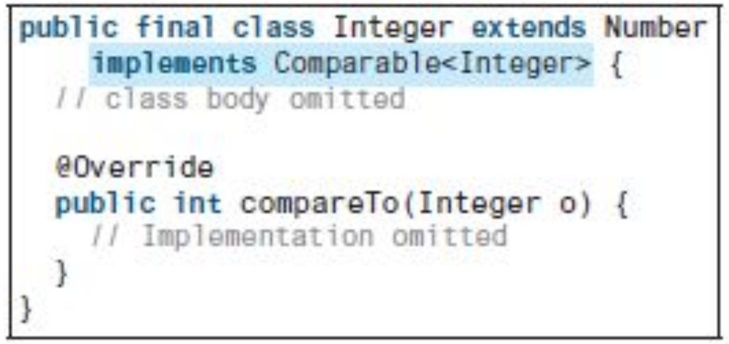
The `Comparable` Interface

- Suppose you want to design a generic method to find the larger/lesser of two objects of the same type, the two objects must be **`comparable`**, so the common behavior for the objects must also be **`comparable`**.
- Java provides the generic **`Comparable`** interface. The generic type E is replaced by a concrete type when implementing this interface.

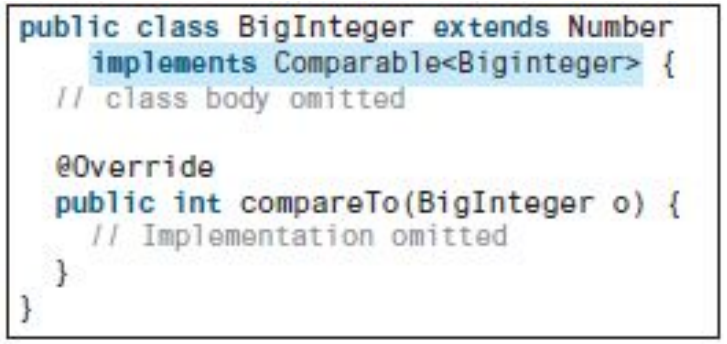
```
// Interface for comparing objects, defined in java.lang
package java.lang;
public interface Comparable<E> {
    public int compareTo(E o);
}
```

The Comparable Interface (2)

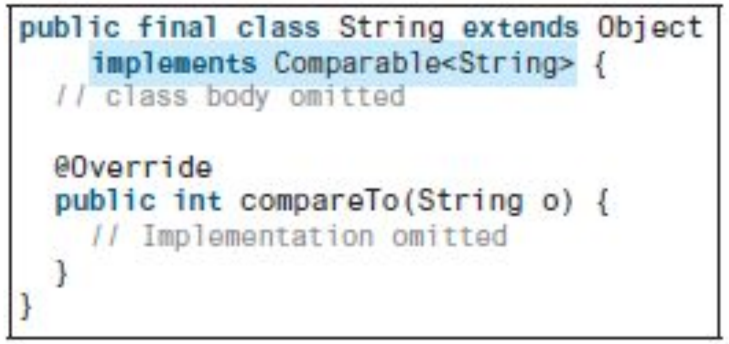
- Many classes in the Java library implement **Comparable** to define a natural order for objects:



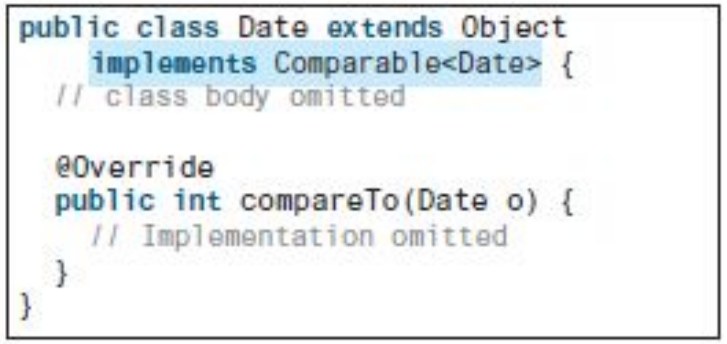
```java
public final class Integer extends Number
    implements Comparable<Integer> {
  // class body omitted

  @Override
  public int compareTo(Integer o) {
    // Implementation omitted
  }
}
```



```java
public class BigInteger extends Number
    implements Comparable<Biginteger> {
  // class body omitted

  @Override
  public int compareTo(BigInteger o) {
    // Implementation omitted
  }
}
```



```java
public final class String extends Object
    implements Comparable<String> {
  // class body omitted

  @Override
  public int compareTo(String o) {
    // Implementation omitted
  }
}
```



```java
public class Date extends Object
    implements Comparable<Date> {
  // class body omitted

  @Override
  public int compareTo(Date o) {
    // Implementation omitted
  }
}
```

compareTo

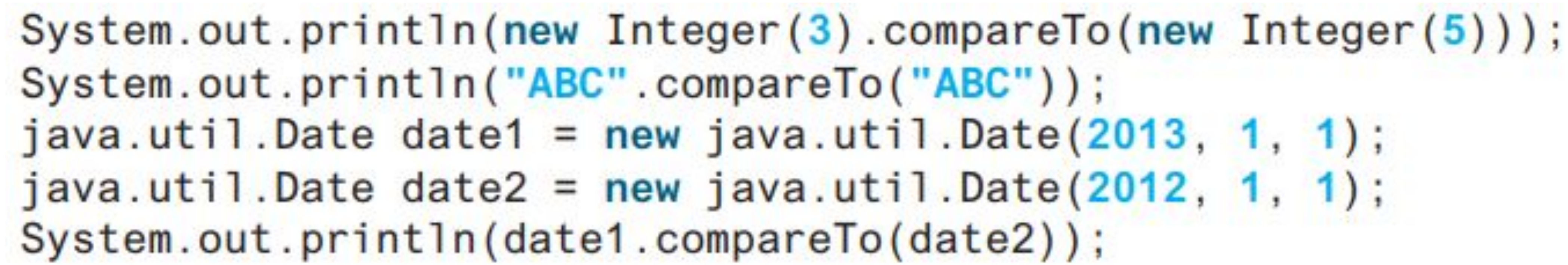
- The **Comparable** interface defines the **compareTo** method for comparing **comparable** objects.
- The **compareTo** method determines the order of this object with the specified object o and returns a negative integer, zero, or a positive integer if this object is less than, equal to, or greater than o.

```
// Interface for comparing objects, defined in java.lang
package java.lang;
public interface Comparable<E> {
    public int compareTo(E o);
}
```

You can also define a class that implements Comparable, with your own definition of compareTo

Example: The `Comparable` Interface

- What is the output?



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

Example: The Comparable Interface

- Since all `Comparable` objects have the `compareTo` method, the `java.util.Arrays.sort(Object[])` method in the Java API uses the `compareTo`, provided the objects are instances of the Comparable interface.

```java
public class ComparableRectangle extends Rectangle
    implements Comparable<ComparableRectangle> {
  /** Construct a ComparableRectangle with specified properties */
  public ComparableRectangle(double width, double height) {
    super(width, height);
  }

  @Override // Implement the compareTo method defined in Comparable
  public int compareTo(ComparableRectangle o) {
    if (getArea() > o.getArea())
      return 1;
    else if (getArea() < o.getArea())
      return -1;
    else
      return 0;
  }
```

```java
public class SortRectangles {
  public static void main(String[] args) {
    ComparableRectangle[] rectangles = {
      new ComparableRectangle(3.4, 5.4),
      new ComparableRectangle(13.24, 55.4),
      new ComparableRectangle(7.4, 35.4),
      new ComparableRectangle(1.4, 25.4)};
    java.util.Arrays.sort(rectangles);
    for (Rectangle rectangle: rectangles) {
      System.out.print(rectangle + " ");
      System.out.println();
    }
  }
}
```

Example: Defining Comparable Implementation

```
public class Mahasiswa implements Comparable<Mahasiswa>{
private long NPM;
....
@override
public int compareTo(Mahasiswa o)
// define your comparison here
}
```

You can also define a class that implements Comparable, with your own definition of compareTo

```
public class Main {
    public static void main(String[] args) {
        MyClass obj1 = new MyClass(10);
        String obj2 = “Hello”;

        int result = obj1.compareTo(obj2);
        System.out.println("Comparison result: " + result);
    }
}

class MyClass implements Comparable {
    private int value;

    public MyClass(int value) {
        this.value = value;
    }

    @Override
    public int compareTo(Object other) {
        MyClass otherObj = (MyClass) other;
        return Integer.compare(this.value, otherObj.value);
    }
}
```

Notice a case where we use **raw type** (e.g. not defining <E>)

The code will compile but produce error

The `Clonable` Interface

- Often, it is desirable to create a copy of an object. To do this, you need to use the clone method and understand the **`Cloneable`** interface.
- The **`Cloneable`** interface specifies that an object can be cloned.
- An interface contains constants and abstract methods, but the **`Cloneable`** interface is a special case the interface is empty.

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

The `Clonable` Interface (2)

- This interface is empty.
- An interface with an empty body is referred to as a marker interface, 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 {
}
```

clone()

- The header for **`clone()`** in **`Object`** is:

```
protected native Object clone() throws CloneNotSupportedException;
```

- The keyword native indicates that this method is not written in Java but is implemented in the JVM for the native platform.
- The keyword protected restricts the method to be accessed in the same package or in a subclass. If you use it, your class must **override** the method and change the visibility modifier to **`public`.**
- Since the clone method implemented for the native platform in the **`Object`** class performs the task of cloning objects, you can invoke the clone method using **`super.clone()`.**

Example

Note: try-catch block
Stay tuned for Exception handling!

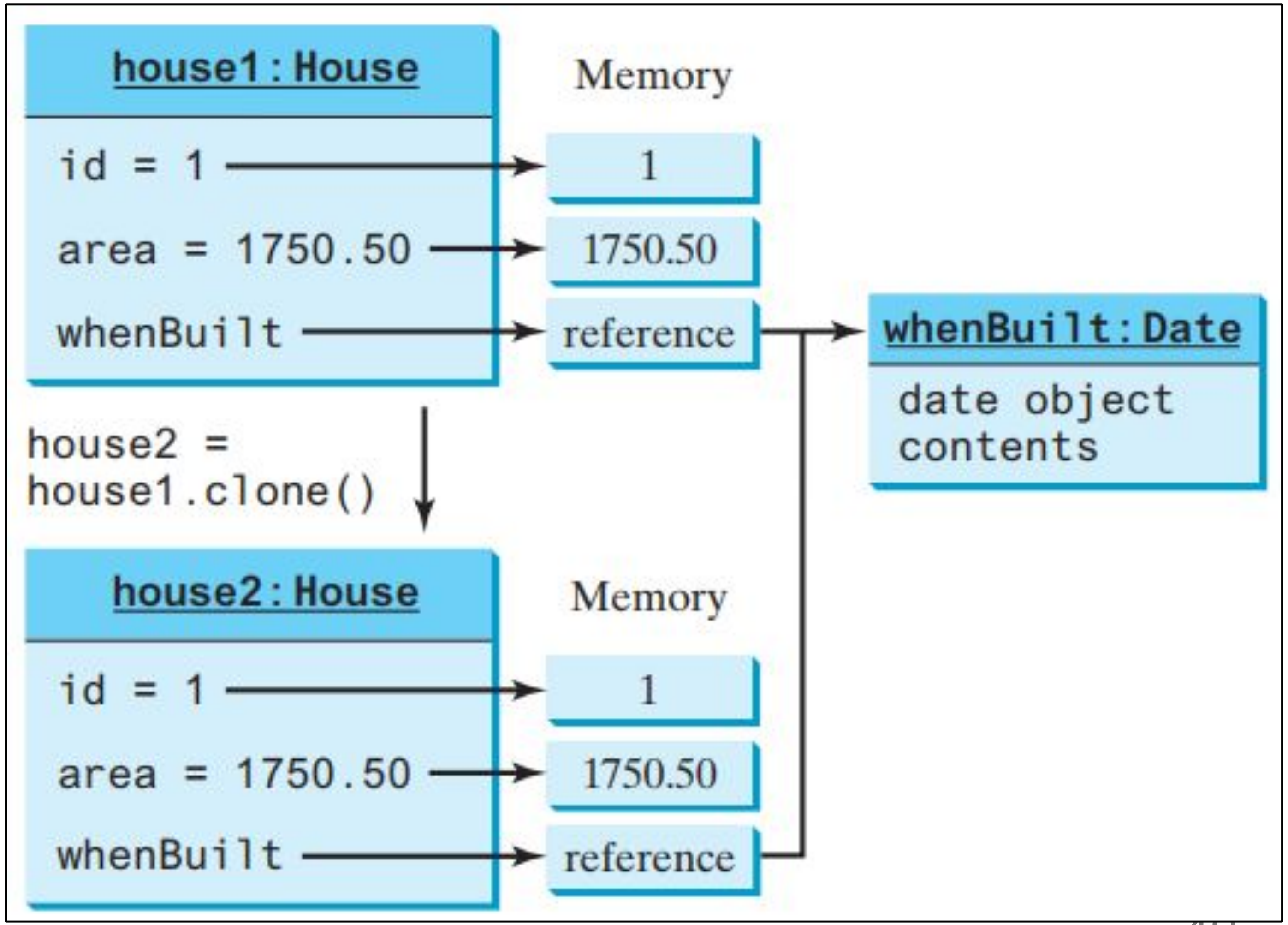
```
public class House implements Cloneable, Comparable<House> {
  private int id;
  private double area;
  private java.util.Date whenBuilt;

  public House(int id, double area) {
    // implememtation
  }

// getters

  @Override /** Override the protected clone method in the Object*/
  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;
  }
}
```

FAKULTAS ILMU KOMPUTER

Shallow vs Deep Copy

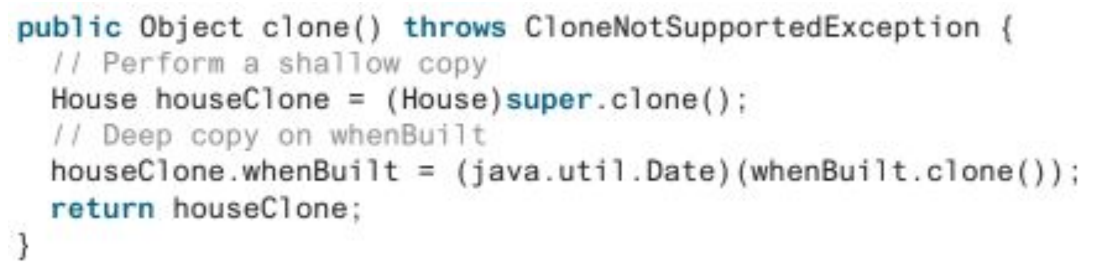
```
House house1 = new House(1, 1750.50);
House house2 = (House)house1.clone();
```

When you create an object of the House and copy it, actually you perform a shallow copy.

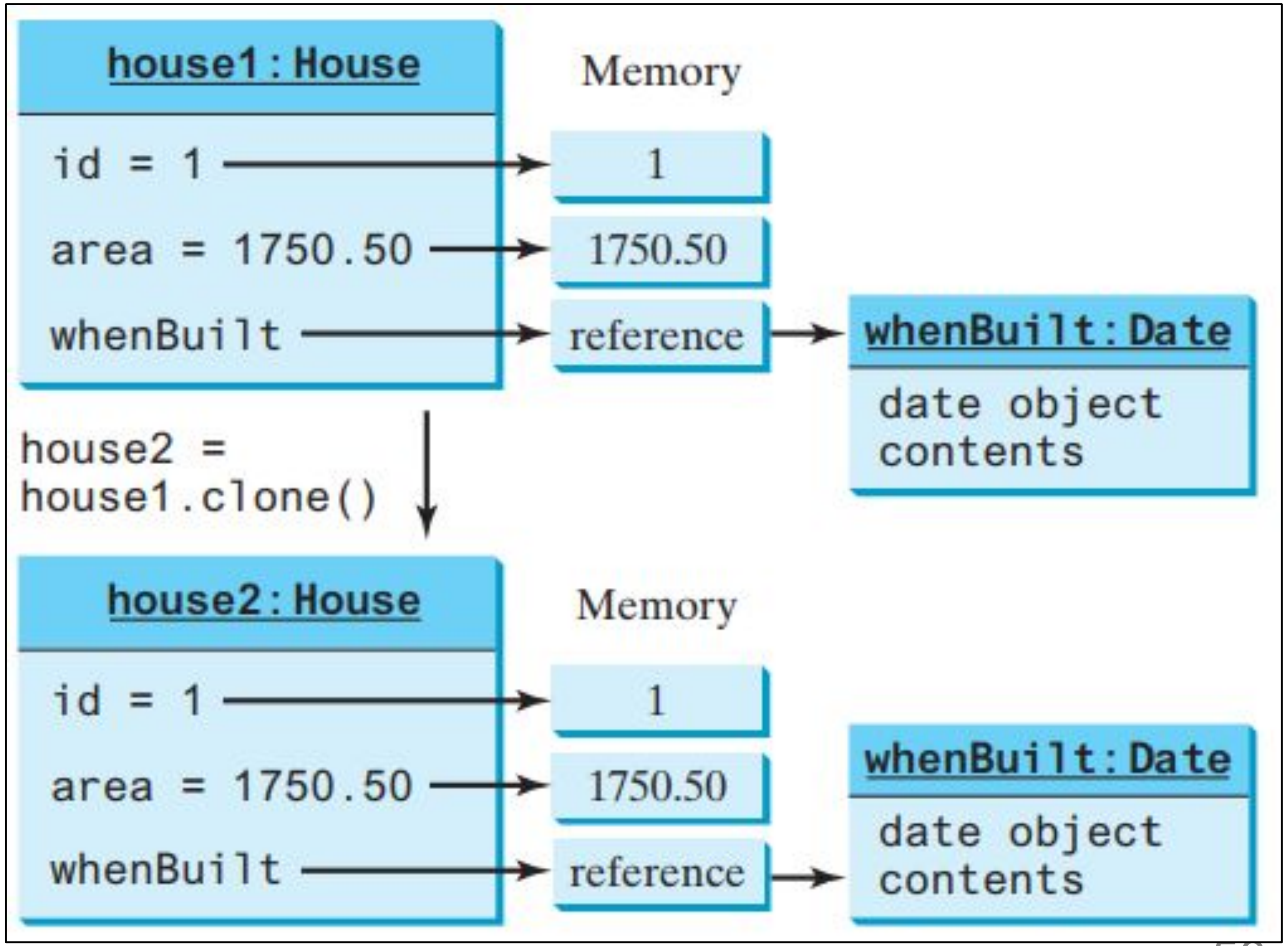


Exercise: The `Cloneable` Interface

Modify the clone() method in House.java to perform a deep copy



```
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;
}
```



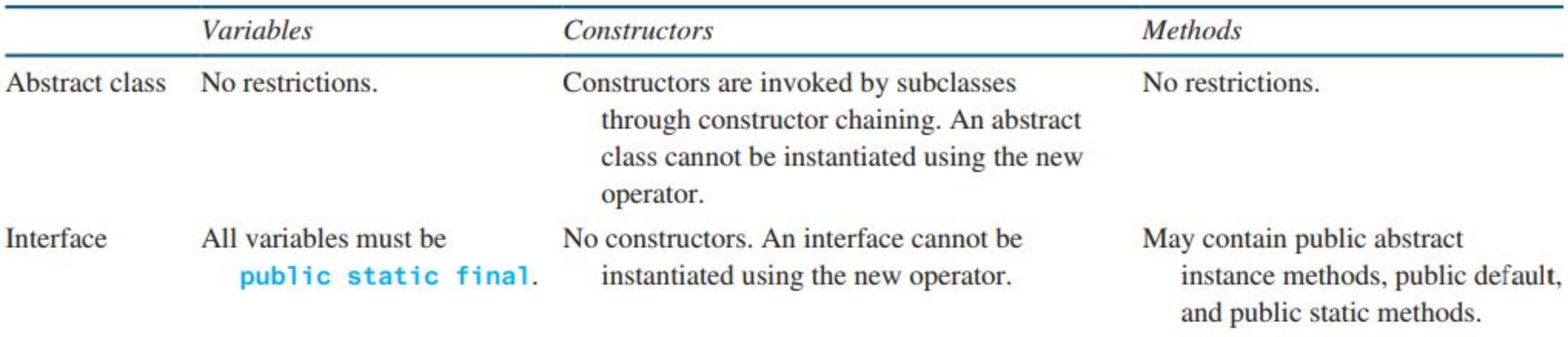
The Serializable Interface

- Stay tuned for I/O topics!

Abstract vs Interface

Variables, Constructors, and Methods

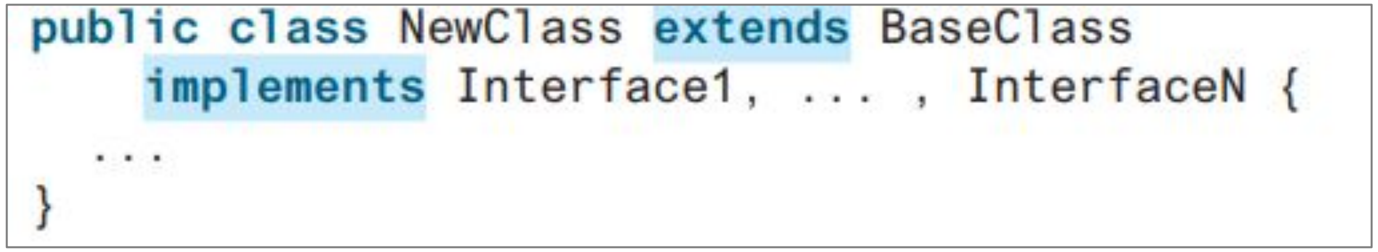
- In an interface, the data must be constants; an abstract class can have all types of data.
- Each method in an interface has only a signature without implementation; an abstract class can have concrete methods.



	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.	May contain public abstract instance methods, public default, and public static methods.

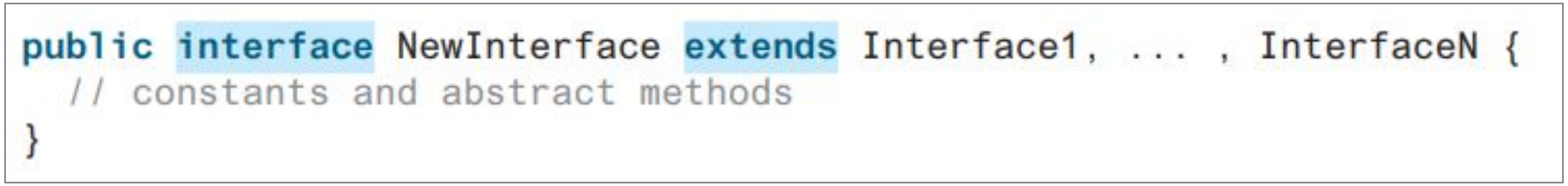
Inheritance of abstract classes and interface

- Java allows only single inheritance but allows multiple extensions



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

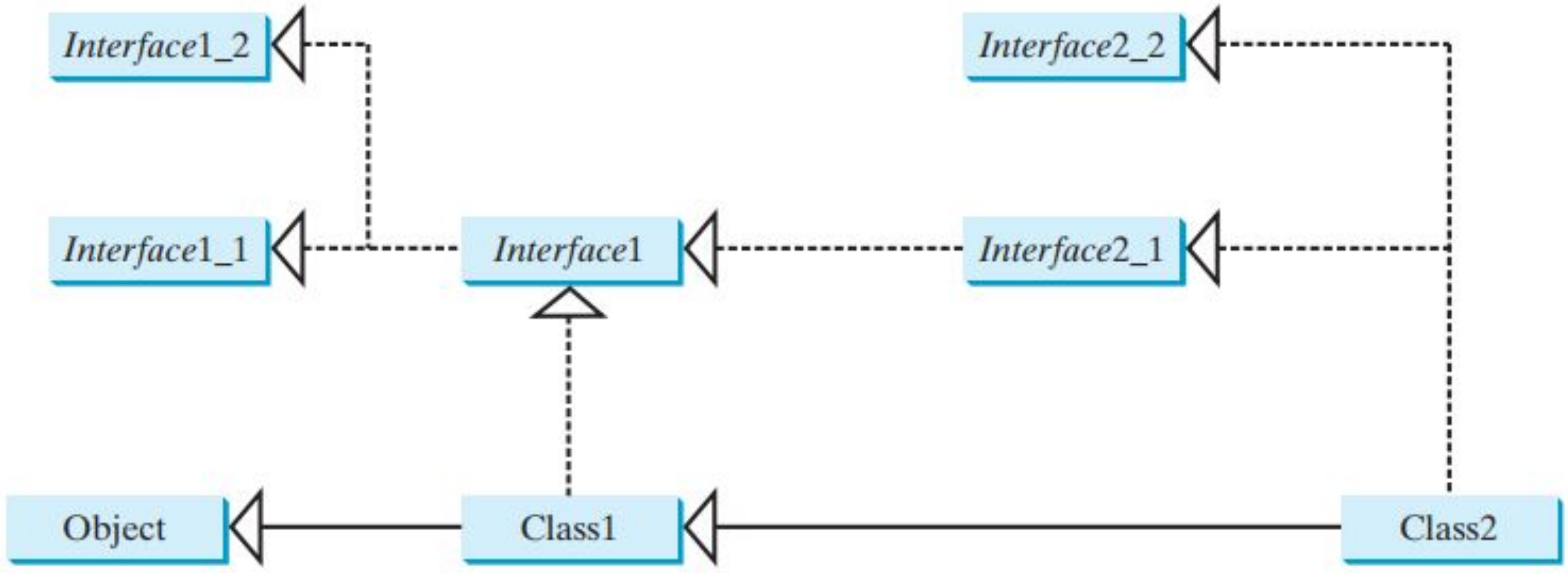
- An interface can inherit other interfaces using the **extends** keyword. Such an interface is called a subinterface.



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

Inheritance of abstract classes and interface (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.



Interface vs Class: When to use what?

- 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.
- A weak **is-a relationship** (or **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.

When to use what?

- 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.

- Use abstract classes or interfaces as **templates, guides or contracts** with default implementations.

Self study

- A detailed comparison and by scenario examples https://ioflood.com/blog/java-abstract-class-vs-interface/
- Advantage, disadvantage, and best practices of Java abstraction https://www.developer.com/java/java-abstraction/
- Cloneable interface http://www.cs.armstrong.edu/liang/intro10e/html/House.html