

A decorative graphic on the left side of the slide, consisting of a network of light blue lines and small circles, resembling a circuit board or a stylized tree structure, extending from the top to the bottom of the frame.

# ABSTRACT CLASS AND INTERFACE

ITX 2001, CSX 3002, IT 2371

# OBJECTIVES

- To design and use abstract and interface class
- To declare interface class to model weak inheritance relationships
- To define a natural order using the Comparable interface
- To know the similarities and differences between an abstract class and an interface class

# EXAMPLE: USING THE GEOMETRICOBJECT CLASS

- This example creates two geometric objects:
  - a circle, and
  - a rectangle,

invokes the `equalArea` method to check if the two objects have equal area, and invokes the `displayGeometricObject` method to display the objects.

TestGeometricObject

# THE ABSTRACT CLASS

- In heritage hierarchy, classes become more specific and concrete with each new subclass.
- If you move from a subclass back up to a superclass, the classes become more general and less specific.
- Class design should be ensured that a superclass contains common features of its subclasses.
- Sometimes a superclass is so abstract that it cannot have specific instances.



# THE ABSTRACT CLASS

- Sometime a superclass is so abstract
  - Which is called “abstract class”
- Add keyword `“abstract”` in front of keyword `“class”`.

# THE ABSTRACT CLASS PROPERTIES

- **The abstract class**
  - Cannot be instantiated
  - Should be extended and implemented in subclasses
- **Test invalid code:**

```
GeometricObject geoObject = new GeometricObject();
```

# THE ABSTRACT MODIFIER

- The abstract class
  - Cannot be instantiated
  - Should be extended and implemented in subclasses
- The abstract method
  - Cannot be implemented in the abstract class because their implementation depends on the specific type of its subclasses.
  - Denoted by placing the keyword “abstract” in the method header.
- In UML notation, the names of abstract classes and methods are *italicized*

# SUPER CLASS

GeometricObject
-color: String -filled: boolean -dateCreated: java.util.Date
+GeometricObject() +getColor(): String +setColor(color: String): void +isFilled(): boolean +setFilled(filled: boolean): void +getDateCreated(): java.util.Date +toString(): String

The color of the object (default: white).

Indicates whether the object is filled with a color (default: false).

The date when the object was created.

Creates a GeometricObject.

Returns the color.

Sets a new color.

Returns the filled property.

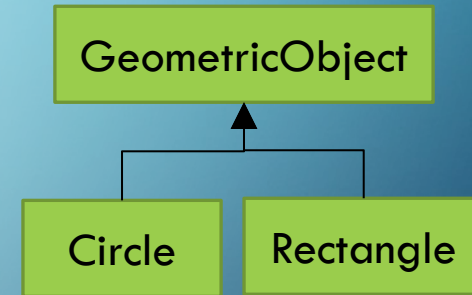
Sets a new filled property.

Returns the dateCreated.

Returns a string representation of this object.

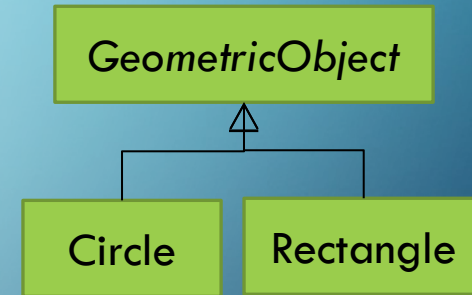
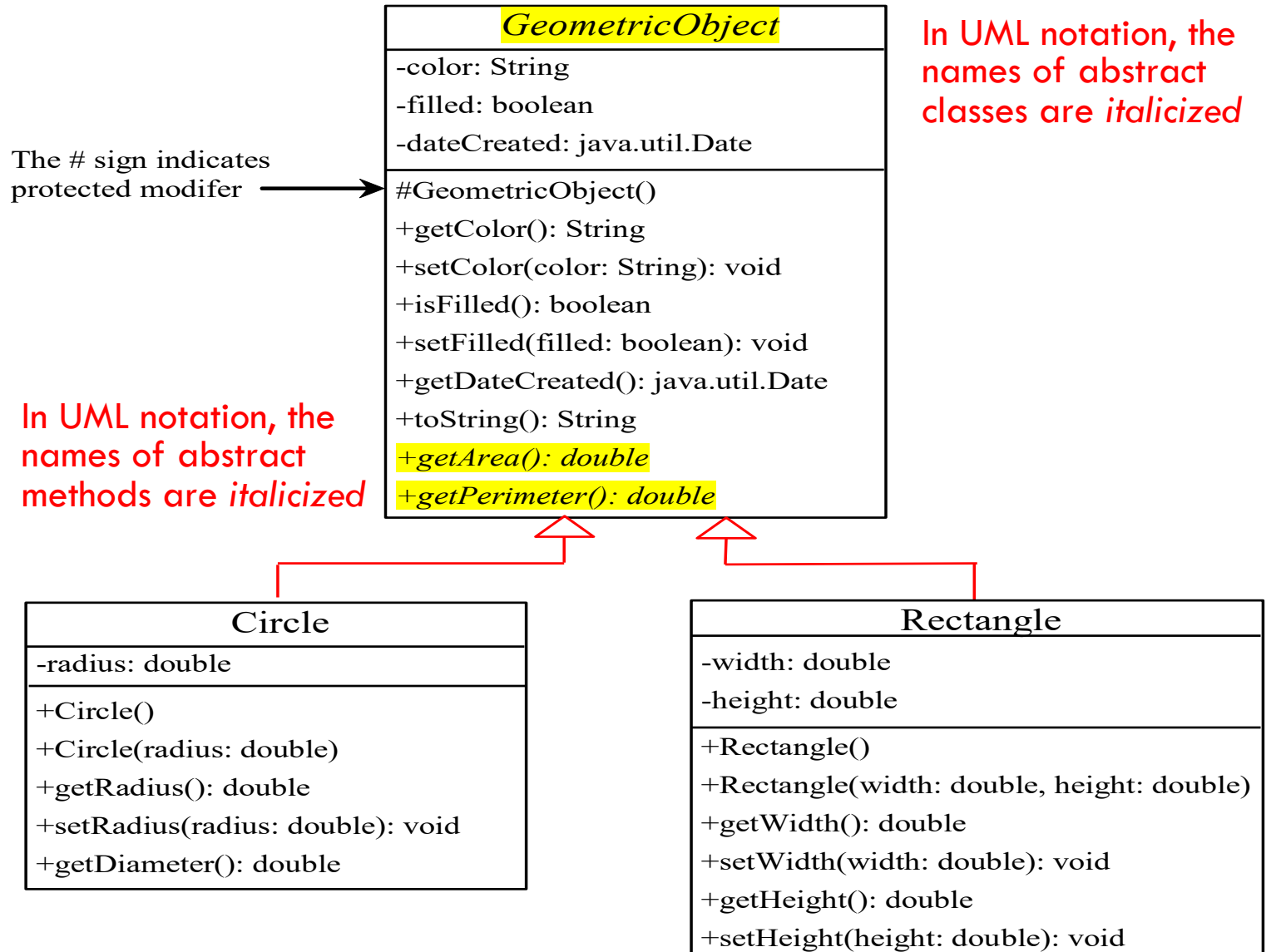
Circle
-radius: double
+Circle() +Circle(radius: double) +getRadius(): double +setRadius(radius: double): void <b>+getArea(): double</b> <b>+getPerimeter(): double</b> +getDiameter(): double

Rectangle
-width: double -height: double
+Rectangle() +Rectangle(width: double, height: double) +getWidth(): double +setWidth(width: double): void +getHeight(): double +setHeight(height: double): void <b>+getArea(): double</b> <b>+getPerimeter(): double</b>





# ABSTRACT CLASS



# ABSTRACT METHOD

- 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 declared 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.

# WHY ABSTRACT CLASS?

- When you want to handle subclasses through an abstract super class.
- For example,
  - Compare area of two objects generated from the different superclasses
    - `equalArea( )` in `TestGeoMetricObject`.
  - Display an geometric object based on its class.
    - `displayGeometricObject( )` in `TestGeometricObject`
- JVM dynamically determines which methods will be invoked at runtime, depending on the specific type of object.

# ABSTRACT CONSTRUCTOR

- 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 CONTAINS NON-ABSTRACT METHODS

- A class that contains abstract methods must be abstract class.
- However, it is possible to declare an abstract class that contains non-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.

## AN ABSTRACT METHOD

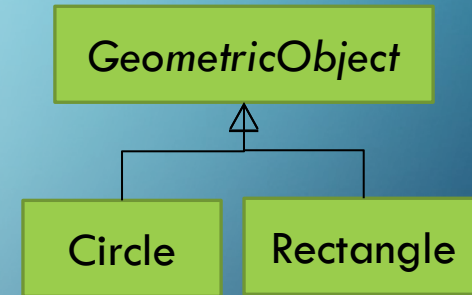
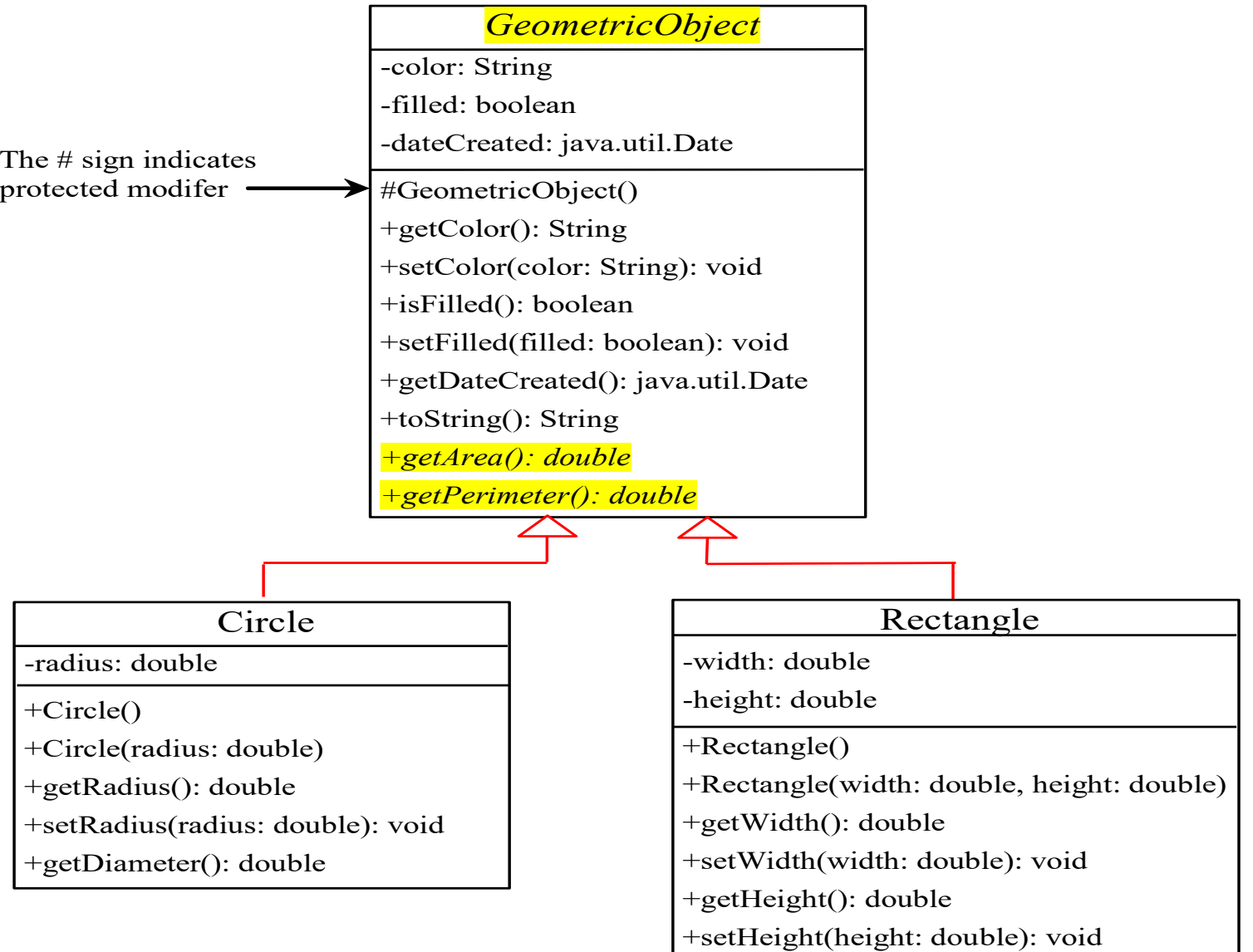
- An abstract method cannot be contained on a non-abstract class
- If a subclass of an abstract superclass does not implement all abstract methods, the subclass must be declared abstract.

# CONCRETE SUPERCLASS & ABSTRACT SUBCLASS

- 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.

# ABSTRACT GEOMETRIC CLASS

The # sign indicates  
protected modifier





# OVERRIDE SUPERCLASS METHOD TO BE ABSTRACT

- A subclass can override a method from its superclass to declare 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 declared abstract.


# ABSTRACT CLASS DATA 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];
```

# THE ABSTRACT CALENDAR CLASS AND ITS GREGORIANCALENDAR SUBCLASS

Optional  
GUI

<i>java.util.Calendar</i>	
#Calendar() +get(field: int): int +set(field: int, value: int): void +set(year: int, month: int, dayOfMonth: int): void +getActualMaximum(field: int): int +add(field: int, amount: int): void +getTime(): java.util.Date +setTime(date: java.util.Date): void	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.
	
<b>java.util.GregorianCalendar</b>	
+GregorianCalendar() +GregorianCalendar(year: int, month: int, dayOfMonth: int) +GregorianCalendar(year: int, month: int, dayOfMonth: int, hour: int, minute: int, second: int)	Constructs a GregorianCalendar for the current time. Constructs a GregorianCalendar for the specified year, month, and day of month. Constructs a GregorianCalendar for the specified year, month, day of month, hour, minute, and second. The month parameter is 0-based, that is, 0 is for January.

# THE ABSTRACT CALENDAR CLASS AND ITS GREGORIANCALENDAR SUBCLASS

Optional  
GUI

- 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

Optional  
GUI

- 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

Optional  
GUI

- The get(int field) method defined in the Calendar class is useful to extract the value for a given time field.
- The time fields are defined as constants such as
  - YEAR,
  - MONTH,
  - DATE,
  - HOURL (for the 12-hour clock),
  - HOURL OF DAY (for the 24-hour clock),
  - MINUTE,
  - SECOND,
  - DAY OF WEEK (the day number within the current week with 1 for Sunday),
  - DAY OF MONTH (same as the DATE value),
  - DAY OF YEAR (the day number within the current year with 1 for the first day of the year),
  - WEEK OF MONTH (the week number within the current month), and
  - WEEK OF YEAR (the week number within the current year). For example, the following code

# INTERFACES

- An *interface* is a class like construct that contains only constants and abstract methods.
- In many ways, an interface is similar to an abstract class,
  - but an abstract class can contain variables and concrete methods as well as constants and abstract methods.
  - But the intent of an interface is to specify common behavior for objects.

# INTERFACES

- For example, using appropriate interfaces, you can specify that objects are comparable, editable, and cloneable.
- To distinguish an interface from a class, Java uses the following syntax to declare an interface:

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



# 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.

# INTERFACE IS A SPECIAL CLASS

- Like an abstract class, you cannot create an instance from an interface using the new operator,
  - but in most case 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.

# DEFINE INTERFACES

- Suppose you want to design a generic method to find the larger of two objects.
  - The objects can be students, dates, or circles.
  - Since compare methods are different for different types of objects, you need to define a generic compare method to determine the order of the two objects.
  - Then you can tailor the method to compare students, dates, or circles.
  - For example, you can use student ID as the key for comparing students, radius as the key for comparing circles, and volume as the key for comparing dates.
- You can use an interface to define a generic compareTo method, as follows:

# INTERFACE EXAMPLE: `TESTEDIBLE.JAVA`

- When a class implements an interface, it implements all the methods defined in the interface with the exact signature and their return type.
- All abstract methods defined in the interface must be overridden.

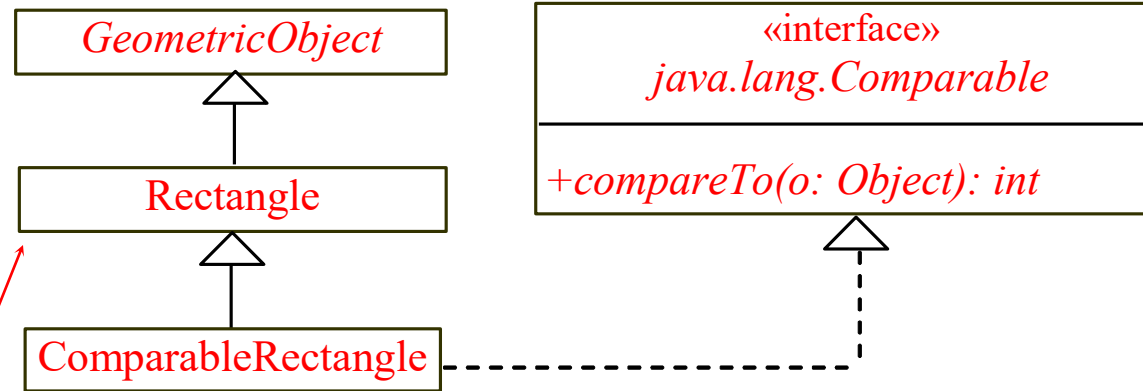
TestEdible



# DECLARING CLASSES TO IMPLEMENT COMPARABLE

*Notation:*

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



You cannot use the `max` method to find the larger of two instances of `Rectangle`, because `Rectangle` does not implement `Comparable`. However, you can declare a new rectangle class that implements `Comparable`. The instances of this new class are comparable. Let this new class be named `ComparableRectangle`.

```
ComparableRectangle rectangle1 = new ComparableRectangle(4, 5);
ComparableRectangle rectangle2 = new ComparableRectangle(3, 6);
System.out.println(Max.max(rectangle1, rectangle2));
```

## INTERFACE EXAMPLE: COMPARABLERECTANGLE.JAVA

- All constants must be declared as static:

```
Public static final returnType CONSTANT_NAME
```

- A constant defined in an interface can be accessed using syntax

```
InterfaceName.CONSTANT_NAME
```

ComparableRectangle

# STRING AND DATE CLASSES

- Many classes (e.g., `String` and `Date`) in the Java library implement `Comparable` to define a natural order for the objects.
- If you examine the source code of these classes, you will see the keyword `implements` used in the classes, as shown below:

```
public class String extends Object  
    implements Comparable {  
    // class body omitted  
}
```

```
public class Date extends Object  
    implements Comparable {  
    // class body omitted  
}
```

```
new String() instanceof String  
new String() instanceof Comparable  
new java.util.Date() instanceof java.util.Date  
new java.util.Date() instanceof Comparable
```

# GENERIC MAX METHOD

```
// Max.java: Find a maximum object
public class Max {
    /** Return the maximum of two objects */
    public static Comparable max
        (Comparable o1, Comparable o2) {
        if (o1.compareTo(o2) > 0)
            return o1;
        else
            return o2;
        }
    }
}
```

(a)

```
// Max.java: Find a maximum object
public class Max {
    /** Return the maximum of two objects */
    public static Object max
        (Object o1, Object o2) {
        if (((Comparable)o1).compareTo(o2) > 0)
            return o1;
        else
            return o2;
        }
    }
}
```

(b)

```
String s1 = "abcdef";
String s2 = "abcdee";
String s3 = (String)Max.max(s1, s2);
```

```
Date d1 = new Date();
Date d2 = new Date();
Date d3 = (Date)Max.max(d1, d2);
```

The return value from the max method is of the Comparable type. So, you need to cast it to String or Date explicitly.



# 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
<b>Abstract class</b>	No restrictions	Constructors are invoked by subclasses through constructor chaining. An abstract class cannot be instantiated using the new operator.	No restrictions.
<b>Interface class</b>	All variables must be <u>public</u> <u>static</u> <u>final</u>	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 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:

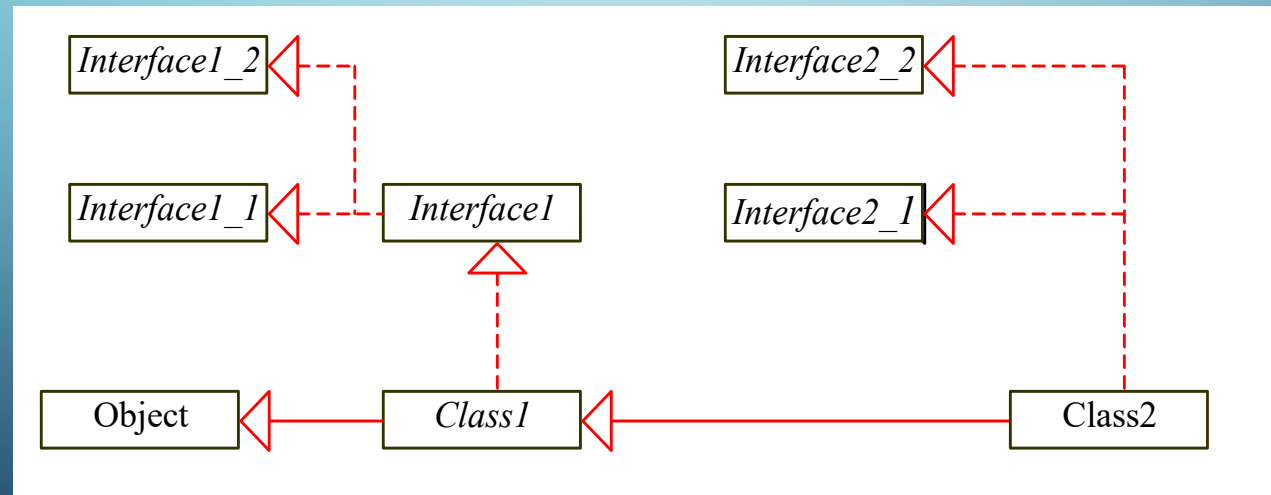
```
public interface T1 {  
    public static final int K = 1;  
  
    public abstract void p();  
}
```

Equivalent

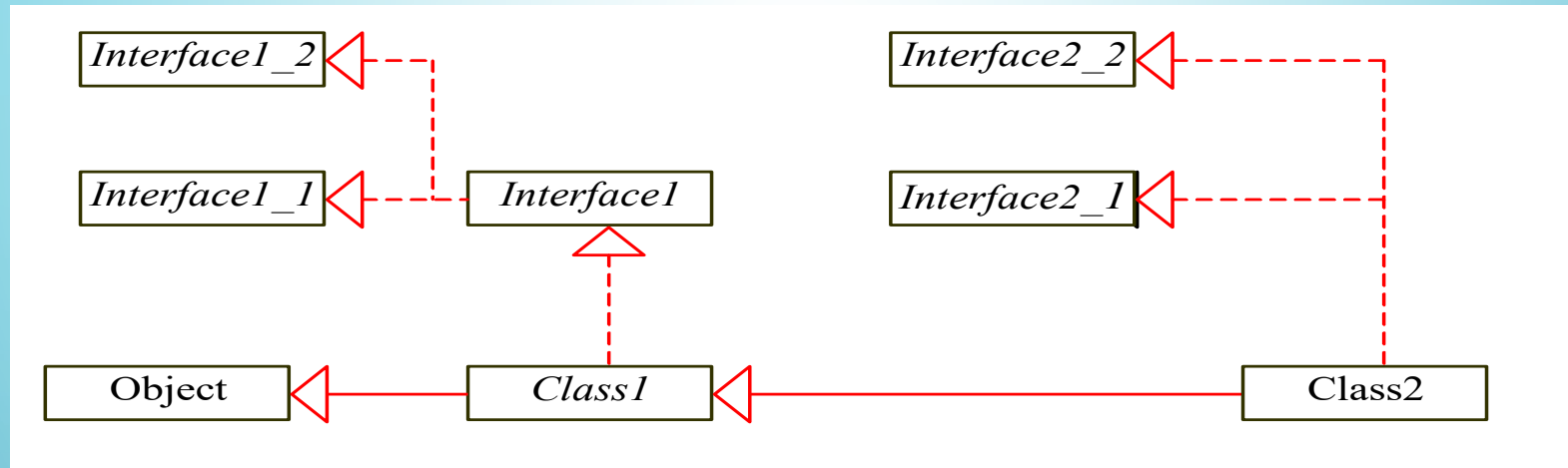
```
public interface T1 {  
    int K = 1;  
  
    void p();  
}
```

# 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.



# INTERFACES VS. ABSTRACT CLASSES, CONT.



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.

# WHETHER TO USE AN INTERFACE OR A CLASS?

Abstract classes and Interfaces can both be used to model common features.

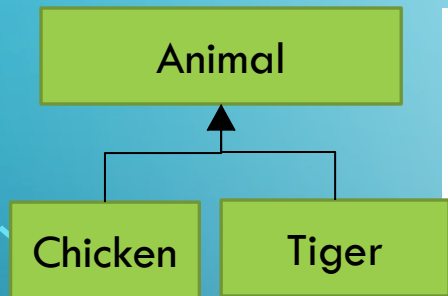
How do you decide whether to use an interface or a class?

- 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. So, their relationship should be modeled using class inheritance.
- A **weak is-a relationship**, also known as an **is-kind-of relationship**, indicates that an object possesses a certain property, 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.

# CREATING CUSTOM INTERFACES

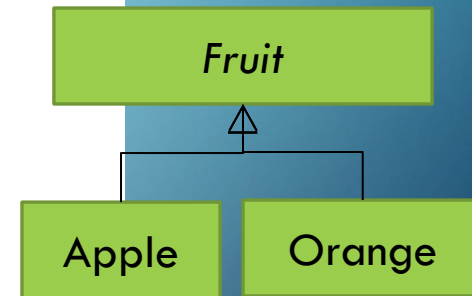
```
public interface Edible {  
    /** Describe how to eat */  
    public String howToEat();  
}
```

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



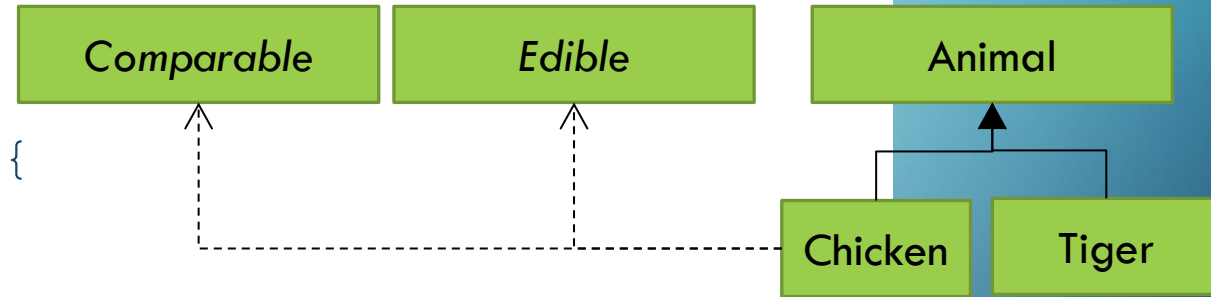
```
class abstract Fruit  
    implements Edible {  
}
```

```
class Apple extends Fruit {  
    public String howToEat() {  
        return "Make apple cider";  
    }  
}  
  
class Orange extends Fruit {  
    public String howToEat() {  
        return "Make orange juice";  
    }  
}
```



# IMPLEMENTS MULTIPLE INTERFACES

```
class Chicken extends Animal implements Edible,  
Comparable {  
    int weight;  
    public Chicken(int weight) {  
        this.weight = weight;  
    }  
    public String howToEat() {  
        return "Fry it";  
    }  
    public int compareTo(Object o) {  
        return weight - ((Chicken)o).weight;  
    }  
}
```

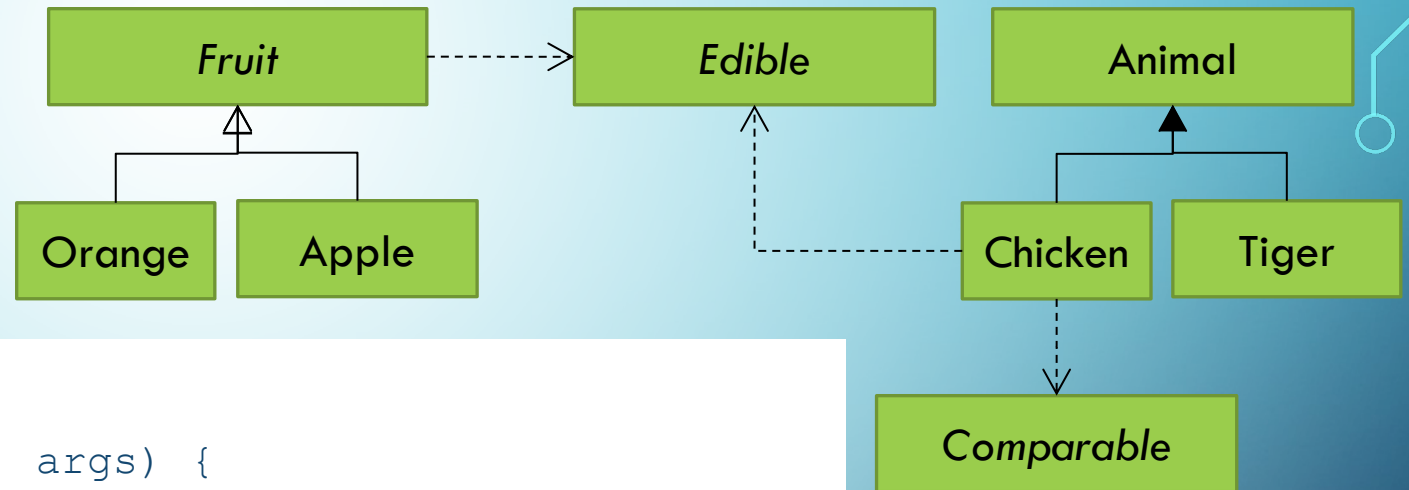




# CREATING CUSTOM INTERFACES, CONT.

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

```
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++)  
            showObject(objects[i]);  
    }  
  
    public static void showObject(Object object) {  
        if (object instanceof Edible)  
            System.out.println(((Edible)object).howToEat());  
    }  
}
```



# THE ACTIONLISTENER INTERFACES

Optional  
GUI

- To respond to a button click, you need to write the code to process the clicking button action.
  - ◆ The button is a source object where the action originates.
- You need to create an object capable of handling the action event on a button.
  - ◆ This object is called a listener.
- Not all objects can be listeners for an action event.

# THE ACTIONLISTENER INTERFACES

Optional  
GUI

- To be a listener, two requirements must be met:
  1. The object must be an instance of the `ActionListener` interface.
    - The `ActionListener` interface contains the `actionPerformed` method for processing the event.
  2. The `ActionListener` object listener must be registered with the source using the method `source.addActionListener(listener)`.

# HANDLING GUI EVENTS

Optional  
GUI

Source object (e.g., button)

Listener object contains a method for processing  
the event.

HandleEvent



# TRACE EXECUTION

Optional  
GUI

```
public class HandleEvent extends JFrame {  
    public HandleEvent() {  
        ...  
        OKListenerClass listener1 = new OKListenerClass();  
        jbtOK.addActionListener(listener1);  
        ...  
    }  
}
```

```
public static void main(String[] args) {  
    ...  
}  
}
```

1. Start from the main method to create a window and display it



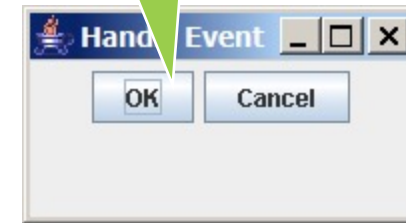
```
class OKListenerClass implements ActionListener {  
    public void actionPerformed(ActionEvent e) {  
        System.out.println("OK button clicked");  
    }  
}
```

# TRACE EXECUTION

Optional  
GUI

```
public class HandleEvent extends JFrame {  
    public HandleEvent() {  
        ...  
        OKListenerClass listener1 = new OKListenerClass();  
        jbtOK.addActionListener(listener1);  
        ...  
    }  
  
    public static void main(String[] args) {  
        ...  
    }  
}  
  
class OKListenerClass implements ActionListener {  
    public void actionPerformed(ActionEvent e) {  
        System.out.println("OK button clicked");  
    }  
}
```

2. Click OK

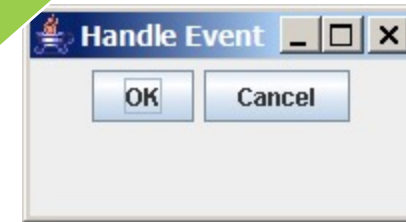


## TRACE EXECUTION

Optional  
GUI

```
public class HandleEvent extends JFrame {  
    public HandleEvent() {  
        ...  
        OKListenerClass listener1 = new OKListenerClass();  
        jbtOK.addActionListener(listener1);  
        ...  
    }  
  
    public static void main(String[] args) {  
        ...  
    }  
}  
  
class OKListenerClass implements ActionListener {  
    public void actionPerformed(ActionEvent e) {  
        System.out.println("OK button clicked");  
    }  
}
```

3. Click OK. The JVM invokes the listener's actionPerformed method





# REFERENCES

[1] Liang, “Introduction to Java Programming”, 6<sup>th</sup> Edition, Pearson Education, Inc.

