Abstract Classes and Interfaces
Motivations

• You have learned how to write simple programs to create and display GUI components. Can you write the code to respond to user actions, such as clicking a button to perform an action?

• In order to write such code, you have to know about interfaces. An interface is for defining common behavior for classes (including unrelated classes). Before discussing interfaces, we introduce a closely related subject: abstract classes.
Abstract Classes and Abstract Methods

Abstract class name is italicized

Methods getArea and getPerimeter are overridden in Circle and Rectangle. Superclass methods are generally omitted in the UML diagram for subclasses.
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;
  }

  /** Get 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();
}
public class Circle extends GeometricObject {
    private double radius;

    public Circle() {
    }

    public Circle(double radius) {
        this.radius = radius;
    }

    /** Return radius */
    public double getRadius() {
        return radius;
    }

    /** Set a new radius */
    public void setRadius(double radius) {
        this.radius = radius;
    }

    @Override /** Return area */
    public double getArea() {
        return radius * radius * Math.PI;
    }

    /** Return diameter */
    public double getDiameter() {
        return 2 * radius;
    }

    @Override /** Return perimeter */
    public double getPerimeter() {
        return 2 * radius * Math.PI;
    }

    /* Print the circle info */
    public void printCircle() {
        System.out.println("The circle is created " + getDateCreated() + " and the radius is " + radius);
    }
}

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

    public Rectangle() {
    }

    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    /** Return width */
    public double getWidth() {
        return width;
    }

    /** Set a new width */
    public void setWidth(double width) {
        this.width = width;
    }

    /** Return height */
    public double getHeight() {
        return height;
    }

    /** Set a new height */
    public void setHeight(double height) {
        this.height = height;
    }

    @Override /** Return area */
    public double getArea() {
        return width * height;
    }

    @Override /** Return perimeter */
    public double getPerimeter() {
        return 2 * (width + height);
    }
}
public class TestGeometricObject {
    /** Main method */
    public static void main(String[] args) {
        // Declare and initialize two geometric objects
        GeometricObject geoObject1 = new Circle(5);
        GeometricObject geoObject2 = new Rectangle(5, 3);

        System.out.println("The two objects have the same area? " +
        equalArea(geoObject1, geoObject2));

        // Display circle
        displayGeometricObject(geoObject1);

        // Display rectangle
        displayGeometricObject(geoObject2);
    }

    /** A method for comparing the areas of two geometric objects */
    public static boolean equalArea(GeometricObject object1,
        GeometricObject object2) {
        return object1/Area() == object2/Area();
    }

    /** A method for displaying a geometric object */
    public static void displayGeometricObject(GeometricObject object) {
        System.out.println();
        System.out.println("The area is " + object/Area());
        System.out.println("The perimeter is " + object/getPerimeter());
    }
}
Abstract method in Abstract class

- An abstract method cannot be contained in a non-abstract class.

- A non-abstract 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.

Example: 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.
A subclass can be abstract even if its superclass is concrete.

**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

An abstract class can be used as a data type.

**Example:** The following statement, which creates an array whose elements are of `GeometricObject` type, is correct.

```java
GeometricObject[] geo = new GeometricObject[10];
```
Two Case Study

• Case Study 1: The Abstract **Number** Class
  – **Number** is an abstract superclass for numeric wrapper classes, BigInteger, and BigDecimal.

• Case Study 2: **Calendar** and **GregorianCalendar**
  – **GregorianCalendar** is a concrete subclass of the abstract class **Calendar**.
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 class-like construct that contains only constants and abstract methods.
• An interface is similar to an abstract class, but the intent of an interface is to specify common behavior for objects.
• Example: you can specify that the objects are
  – comparable,
  – edible,
  – cloneable using appropriate interfaces.
Define an Interface

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

```java
modifier interface InterfaceName {
    /** Constant declarations */
    /** Abstract method signatures */
}
```

**Example:**

```java
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 byte-code 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.
Example

- The relationship between the class and the interface is known as *interface inheritance*.
- Interface inheritance is accomplished by letting the class for the object implement this interface using the *implements* keyword.
- When a class implements an interface, it implements all the methods defined in the interface with the exact signature and return type.
- **Example:** the classes *Chicken* and *Fruit* implement the *Edible* interface.
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());
            if (objects[i] instanceof Animal) {
                System.out.println(((Animal)objects[i]).sound());
            }
        }
    }
}

abstract class Animal {
    /** Return animal sound */
    public abstract String sound();
}

class Chicken extends Animal implements Edible {
    @Override
    public String howToEat() {
        return "Chicken: Fry it";
    }
    @Override
    public String sound() {
        return "Chicken: cock-a-doodle-doo";
    }
}

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

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

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

class Orange extends Fruit {
    @Override
    public String howToEat() {
        return "Orange: Make orange juice";
    }
}
```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());
            }
            if (objects[i] instanceof Animal) {
                System.out.println(((Animal)objects[i]).sound());
            }
        }
    }
}

abstract class Animal {
    /** Return animal sound */
    public abstract String sound();
}

class Chicken extends Animal {
    @Override
    public String howToEat() {
        return "Chicken: Fry it";
    }
    @Override
    public String sound() {
        return "Chicken: cock-a-doodle-doo";
    }
}

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

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

A constant defined in an interface can be accessed using syntax `InterfaceName.CONSTANT_NAME`
The Comparable Interface

- Suppose you want to compare two objects of the same type, such as two students, two dates, two circles, … etc.
- The two objects must be comparable, so the common behavior for the objects must be comparable
- Java provides the Comparable interface which defines the compareTo method for comparing objects.

```java
// Interface for comparing objects, defined in java.lang
package java.lang;

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

returns a negative integer, zero, or a positive integer if this object is less than, equal to, or greater than o.
The Comparable Interface

- The Comparable interface is a generic interface. The generic type E is replaced by a concrete type when implementing this interface.
- Many classes in the Java library implement Comparable to define a natural order for objects.
- Examples of the Integer, BigInteger, String, and Date classes from the Java API.
The Comparable Interface

- Let $n$ be an Integer object,

- $s$ be a String object,

- and $d$ be a Date object.
Example of `compareTo` use

```java
1  System.out.println(new Integer(3).compareTo(new Integer(5)));
2  System.out.println("ABC".compareTo("ABE"));
3  java.util.Date date1 = new java.util.Date(2013, 1, 1);
4  java.util.Date date2 = new java.util.Date(2012, 1, 1);
5  System.out.println(date1.compareTo(date2));
```

Output

```
-1
-2
1
```
Generic sort Method

• The `java.util.Arrays.sort(Object[])` method requires that the elements in an array are instances of `Comparable<E>`

• The method in the Java API uses the `compareTo` method to compare and sorts the objects in an array
import java.math.*;

public class SortComparableObjects {
    public static void main(String[] args) {
        String[] cities = {"Savannah", "Boston", "Atlanta", "Tampa"};
        java.util.Arrays.sort(cities);
        for (String city : cities) {
            System.out.print(city + " ");
        }
        System.out.println();

        BigInteger[] hugeNumbers = {new BigInteger("2323231092923992"),
                                     new BigInteger("432232323239292"),
                                     new BigInteger("54623239292")};
        java.util.Arrays.sort(hugeNumbers);
        for (BigInteger number : hugeNumbers) {
            System.out.print(number + " ");
        }
    }
}
import java.math.*;

public class SortComparableObjects {
    public static void main(String[] args) {
        String[] cities = {"Savannah", "Boston", "Atlanta", "Tampa"};
        java.util.Arrays.sort(cities);
        for (String city : cities)
            System.out.print(city + " ");
        System.out.println();

        BigInteger[] hugeNumbers = {new BigInteger("2323231092923992"),
                                    new BigInteger("432232323239292"),
                                    new BigInteger("54623239292")};
        java.util.Arrays.sort(hugeNumbers);
        for (BigInteger number : hugeNumbers)
            System.out.print(number + " ");
    }
}
Defining Classes to Implement Comparable

- You cannot use the `sort` method to sort an array of `Rectangle` objects, because `Rectangle` does not implement `Comparable`.
- However, you can define a new rectangle class that implements `Comparable`. The instances of this new class are comparable.
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;
    }

    @Override // Implement the toString method in GeometricObject
    public String toString() {
        return "Width: " + getWidth() + " Height: " + getHeight() + " Area: " + getArea();
    }
}

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();
        }
    }
}
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;
        }
    }

    @Override
    public String toString() {
        return " Width: \$\text{%.1f}\$ Height: \$\text{%.1f}\$ Area: \$\text{%.1f}\$
    }
}

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();
    }
}
Benefits of Interface

• An interface provides another form of **generic programming**.
• It would be difficult to use a generic **sort** method to sort the objects **without** using an interface in the previous example.
• Java does not provide multiple inheritance, but with interface inheritance we were able to inherit **Comparable** and **Rectangle** at the same time.
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.

```java
package java.lang;

public interface Cloneable {
}
```
The Cloneable Interfaces

- Many classes in the Java library (e.g., Date, Calendar, and ArrayList) implement Cloneable. Thus, the instances of these classes can be cloned.

- To define a custom class that implements the Cloneable interface, the class must override the clone() method in the Object class.
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;
        }
    }

    public int compareTo(House o) {
        if (area > o.area)
            return 1;
        else if (area < o.area)
            return -1;
        else
            return 0;
    }
}

Dr. Rania Khairy
Software Engineering and Development Tool
The clone method

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

- the clone method defined in the Object class.
- 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.
The clone method

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

- The clone method in the Object class copies each field from the original object to the target object.
- If the field is of a primitive type, its value is copied. For example, the value of area (double type) is copied from house1 to house2.
- If the field is of an object, the reference of the field is copied.
Shallow copy

```python
house1: House
  id = 1
  area = 1750.50
  whenBuilt

house2 = house1.clone()

house2: House
  id = 1
  area = 1750.50
  whenBuilt

house1.whenBuilt == house2.whenBuilt is true.
```
Deep Copy

```java
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;
}
```

```
house1: House
id = 1
area = 1750.50
whenBuilt

house2: House
id = 1
area = 1750.50
whenBuilt

Memory

1
1750.50
reference

whenBuilt: Date
date object contents

house1.whenBuilt == house2.whenBuilt will be false.
```
# Interfaces vs. Abstract Classes

<table>
<thead>
<tr>
<th>Variables</th>
<th>Constructors</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract class</td>
<td>No restrictions.</td>
<td>Constructors are invoked by subclasses through constructor chaining. An abstract class cannot be instantiated using the new operator.</td>
</tr>
<tr>
<td>Interface</td>
<td>All variables must be <strong>public static final</strong>.</td>
<td>No constructors. An interface cannot be instantiated using the new operator.</td>
</tr>
</tbody>
</table>
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?**
- A **strong is-a relationship** that clearly describes a parent-child relationship should be modeled using classes.
- A **weak is-a relationship**, known as **is-kind-of relationship** can be modeled using interfaces.
- You can also use interfaces to circumvent single inheritance restriction if multiple inheritance is desired.
Class Design Guidelines

• coherent purpose
  – A class should describe a single entity. You can use a class for students, for example, but you should not combine students and staff in the same class, because students and staff have different entities.

• separate responsibilities
  – A single entity with too many responsibilities can be broken into several classes.
  – The classes String, StringBuilder, and StringBuffer all deal with strings, for example. The String class deals with immutable strings, the StringBuilder class is for creating mutable strings, and the StringBuffer class is similar to StringBuilder except that StringBuffer contains synchronized methods for updating strings.
Class Design Guidelines

• **naming conventions**
  – Follow standard Java programming style and naming conventions.
  – Choose informative names for classes, data fields, and methods. A popular style is to place the data declaration before the constructor and place constructors before methods.

• **no-arg constructor**
  – In general, you should consistently provide a public no-arg constructor for constructing a default instance.
  – If you want to prevent users from creating an object for a class, you can declare a private constructor in the class, as is the case for the `Math` class.
Class Design Guidelines

• **encapsulate data fields**
  – use the private modifier to hide its data from direct access by clients. This makes the class easy to maintain.
  – Provide a getter method only if you want the data field to be readable, and provide a setter method only if you want the data field to be updateable

• **Instance vs Static**
  – A variable or method that is dependent on a specific instance of the class must be an instance variable or method.
  – A variable that is shared by all the instances of a class should be declared static.

• **Override**
  – override the equals method and toString method defined in the Object class whenever possible and needed.
Class Design Guidelines

• Inheritance vs. Aggregation
  – The difference between inheritance and aggregation is the difference between an is-a and a has-a relationship. For example, an apple is a fruit; however a person has a name.

• independent properties
  – You should not declare a data field that can be derived from other data fields. For example, a Person class has two data fields: birthDate and age. Since age can be derived from birthDate, age should not be declared as a data field.