Abstract Classes and Interfaces
Overview: Class Types

From Least to Most Restricted:

1. Concrete Classes (e.g. public class MyClass)
   - Data members: static (class) and instance (object)
   - Methods: static and instance
   - Can Create Instances: Yes (‘concrete’)

2. Abstract Classes (e.g. public abstract class MyAClass)
   - Data members: static and instance
   - Methods: static and instance – at least one instance method is abstract (i.e. has signature, no body)
   - Can Create Instances: No

3. Interfaces (e.g. public interface MyInterface)
   - Data members: public static final constants only
   - Methods: only public abstract instance methods
   - Can Create Instances: No
Abstract Method

Definition
A method which has a signature, but no body. All abstract methods are instance methods (non-static).

• e.g. public abstract int deviseNumber();

Purpose
Class design: permits defining a method signature whose definition may be provided in subclasses.
Abstract Class

Definition
– A class which may not have any instances created from it, used only as a template for subclasses.
  – Otherwise, it is a normal class, and is included in the class inheritance hierarchy.
– All classes containing abstract methods must be declared abstract.
– e.g. public abstract class GeometricObject() { … }

Class Design
– In general, superclasses should be designed to contain common data and methods of subclasses (to maximize code reuse, e.g. Object class)
– Defines a common reference type for these (possibly very) different subclasses
  – e.g. GeometricObject g = new Circle(1.0);
Constructors for Abstract Classes
Provide means to initialize instance data defined in the class
*(protected access more appropriate than public)*

Subclasses of Abstract Classes
Must implement all abstract methods, or also be declared abstract
The GeometricObject class is the superclass for Circle and Rectangle.

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.
The new \textit{GeometricObject} class contains abstract methods.
Examples

TestGeometricObject.java

Note
Abstract GeometricObject class allows us to get areas and perimeters of Circle and Rectangle Objects using a single reference type.
A Yet More Restricted Class Type: Interfaces in Java

Definition

– A type of class which defines only \textit{(public static final)} constants and \textit{(public)} abstract \textit{instance} methods.
– Provides a reference type from which the \textit{interface} may be used to act on objects associated with the interface
– \textbf{NOTE:} Interfaces are \textit{not} part of the class hierarchy

Java Syntax

Defined using the “interface” rather than “class” keyword

• e.g. public \texttt{interface} Cloneable { ... }
Motivation for Interfaces in Java

**Multiple Inheritance is Prohibited**

We cannot inherit from multiple classes; in particular, state is only inherited through a strict linear path in the inheritance tree (hierarchy)

**But...**

– Want different data types to have common methods to support generic programming (e.g. the ability to compare objects using a single interface (Comparable))

– A class may ‘implement’ one or more interfaces to support these ‘generic’ types of computation.
Example: Comparable Interface

Purpose

- Allow definition of a method for determining which of a pair of objects of the same class is ‘larger’ or if they are the ‘same size.’
- Can then use compareTo() to compare Strings, Students (e.g. by student id), Geometric objects (e.g. by area), etc. using a single method with different definitions (one per class)

```java
package java.lang

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

Returns:
-1: this < argument
0: this, argument same
1: this > argument
public class String extends Object implements Comparable {
    ... }

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

**Interface as a Reference Variable Type**

The following are valid for String object s and Date object d:

- s instanceof String,
- s instanceof Object,
- s instanceof Comparable
- d instanceof java.util.Date,
- d instanceof Object,
- d instanceof Comparable
public class Max {
    public static Comparable max(Comparable o1, Comparable o2) {
        if (o1.compareTo(o2) > 0)
            return o1;
        else
            return o2
    }
}

Example Usage:
String s1 = “a”; String s2 = “b”;
String s3 = (String)Max.max(s1,s2);
Date d1 = new Date(); Date d2 = new Date();
Date d3= (Date)Max.max(d1,d2);

Objects of any class that implements Comparable can be used with Max.max() (e.g. a revised Rectangle class)
Interfaces and Inheritance

Classes Implementing Interfaces

– Concrete and Abstract classes may inherit from only one parent.

– However, they may implement multiple interfaces.
  
  public class A extends B implements InterfaceA, InterfaceB, ..., InterfaceN { }

Interfaces extending Interfaces

Interfaces may inherit from and extend one or more interfaces.

  public Interface NewInterface extends IntA, ...., IntN { };
Objects of Class2 are instances of *all* the other classes and interfaces shown. This means variables referring to a Class2 object may be any of these types.
**Marker Interface**

- An interface that contains no constants or methods; ‘flags’ a class as having certain properties (e.g. to tell Java to permit certain operations)
- e.g. “Cloneable” (designates that objects of a class may be copied)

```java
public class House implements Cloneable, Comparable {
    ... 
}
```

```java
House h1 = new House(1,1750.50); // id, area
House h2 = (House)housel.clone();
```

.. See text for details.

**Shallow vs. Deep Copies**

- Shallow: object references copied by value (copies reference to a single object) - danger of manipulating the “original” data in this case
- Deep: object data is copied into new objects, and “copied” references point to the new objects and not the original ones (e.g. using clone())