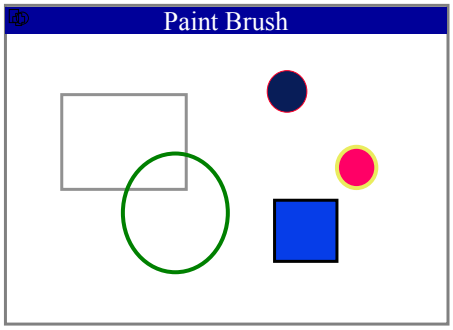
ADTs in Java: Abstract Classes and Interfaces

Abstract Classes

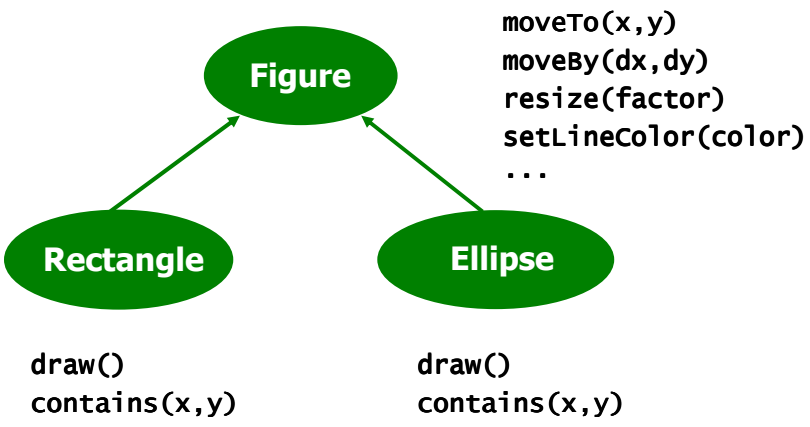
- An abstract class cannot be instantiated
  - It is used in a class hierarchy to organize common features at appropriate levels
- An abstract method has no implementation, just a name and signature
  - An abstract class often contains abstract methods
  - An abstract class is often too generic to be of use by itself
- Abstract methods can appear in classes that themselves been declared `abstract`
  - The modifier **`abstract`** is used to define abstract classes & methods
  - The children of the abstract class are expected to define implementations for the abstract methods in ways appropriate for them
  - If a child class does not define all abstract methods of the parent, then the child is also abstract

Abstract Classes

- Consider the drawing of a paint-brush application:
  - We want to represent the drawing as a collection of objects, each represents a certain figure
  - We want to represent each type of figure with a suitable class. Objects of these classes will know how to draw themselves
  - The classes have common properties (`location`, `size`, `color`, ...) and common behaviors (`moveTo()`, `resize()`, ...).
  - We want to have a common superclass for all these classes which will define all the common properties and behaviors



Abstract Classes



Abstract Classes

```
/**
 * A geometrical figure. A Figure has a location and size
 * it can draw itself on the screen ...
 */
public class Figure {

    // The top-left corner of the rectangular area that
    // contains the figure
    protected int x,y;

    // The dimension of the figure (of the rectangular
    // area that contains the figure)
    protected int width, height;

    // The color of the border of the figure
    private Color lineColor;

    // ... other properties
```


Abstract Classes

```
/**
 * Constructs a new Figure with given location and dimension
 * @param x,y The location of the figure
 * @param width, height The dimensions of the figure
 */
public Figure(int x, int y, int width, int height) {
     this.x = x;
     this.y = y;
     this.width = width;
     this.height = height;
    }
/**
 * Returns the color of the border of this figure
 * @return The color of the border of this figure
 */
public Color getLineColor() {
   return lineColor;
}
```

Abstract Classes

```
/**
 * Moves the figure to a new location
 * @param x,y The coordinates of the new location
 * The top left corner of the rectangle that will
 * contain the figure in its new location
 */
public void moveTo(int x, int y) {
   this.x = x;
   this.y = y;
}
/**
 * Moves the figure by a given offset.
 * @param dx,dy The given offset.
 */
public void moveBy(int dx, int dy) {
    moveTo(x+dx, y+dy);
}
```

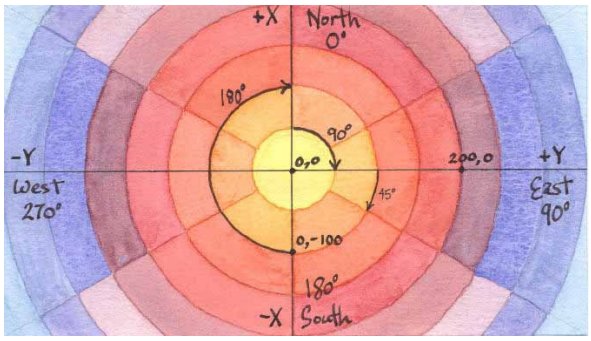

Abstract Classes

```
/**
 * A figure of a rectangle on the plane ...
 */
public class Rectangle extends Figure {

/**
 * Constructs a new Rectangle with given location and
 * dimensions.
 * @param x,y The top-left corner of the rectangle
 * @param width, height The dimensions of the rectangle
 */
public Rectangle(int x, int y, int width, int height) {
     super(x, y, width, height);
}
```

Abstract Classes

```
/**
 * Draws this rectangle
 */
public void draw() {
  Turtle painter = new Turtle();
  painter.setLocation(x,y);
  painter.setAngle(0);
  painter.tailDown();
  painter.moveForwards(width);
  painter.turnRight(90);
  painter.moveForwards(height);
  painter.turnRight(90);
  painter.moveForwards(width);
  painter.turnRight(90);
  painter.moveForwards(height);
  painter.hide();
}
```


Turtle Graphics in a Java Program



- The center of the graphics window is turtle location 0,0. Positive X is to the right; positive Y is up. Headings (angles) are measured in degrees clockwise from the positive Y axis.
  - This differs from the common mathematical convention of measuring angles counterclockwise from the positive X axis
- The turtle is represented as an isosceles triangle; the actual turtle position is at the midpoint of the base (the short side)

Abstract Classes

- We've used class **Figure** to ease our design
  - It is also correct from the point of view of our abstraction
- However, note that we will never want to create instances of class **Figure**, only of subclasses of this class!
  - To denote this, we define the class **Figure** as an **abstract class**

```
/**
 * A geometrical figure. A Figure has a location and size
 * it can draw itself on the screen ...
 */
public abstract class Figure {
    // The top-left corner of the rectangular area that
    // contains the figure
    protected int x,y;
    // ...
```

- Now, we cannot create instances of class **Figure**, only instances of subclasses of **Figure** which are not defined abstract

Abstract Classes

- What are the benefits of defining a class abstract?
  - Note that the **draw()** method appears in the interface of all subclasses of class **Figure**, but is implemented differently in each one of them
- If we have defined this method in class **Figure**, it was to say that every figure can draw itself
  - We would then override this method in every specific subclass and could draw an heterogeneous collection of figures with a single loop

Abstract Classes

```
/**
 * A picture made of geometrical figures
 */
public class PaintBrushPicture {
    private Vector figures;
    /**
     * Constructs an empty picture
    */
    public PaintBrushPicture() {
        figures = new Vector();
    }
    /**
     * Adds a new figure to the picture
     */
    public void add(Figure figure) {
        figures.addElement(figure);
    }
```


Abstract Classes

```
/**
 * Paints this picture
 */
public void draw() {
   for (int i=0; i<figures.size(); i++) {
         ((Figure)figures.elementAt(i)).draw();
      }
 }
 ...
}
```

Abstract Classes

- But how should we define the method `draw()` in class `Figure`?
  - There is no meaning for drawing an abstract figure!
- We can make a workaround and write an empty implementation for `draw()` in `Figure`
  - This is NOT clean! It is a patch!
  - Moreover what about methods like `contains()` ?
- The catch is that we do not have to implement this method, because no one can create instances of class `Figure`!
  - Instead we declare the method as abstract and omit its body

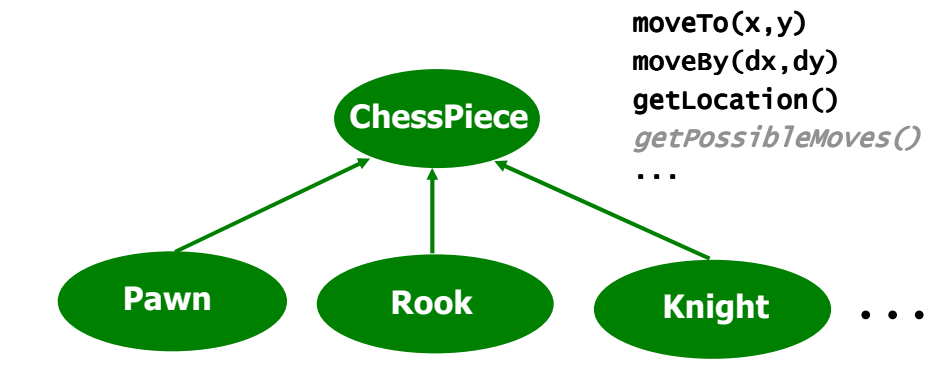
Abstract Classes

```
/**
 * A geometrical figure. ...
 */
public abstract class Figure {
    ... state variables as before
/**
 * Draws this figure
 */
public abstract void draw();
/**
 * Checks a given point is in the interior of
 * this figure.
 */
public abstract boolean contains(int x, int y);
```

Abstract Classes

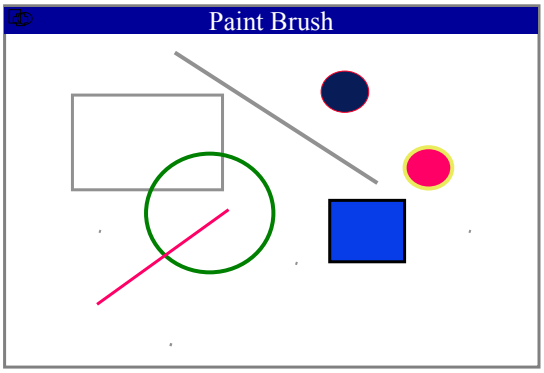
- Now, a subclass of class **`Figure`** must either be abstract or implement all the abstract methods that are defined in class **Figure**
- An abstract method cannot be declared as **`final`**, because it must be overridden in a child class
- An abstract method cannot be declared as **`static`**, because it cannot be invoked without an implementation
- Abstract classes are placeholders that help organize information and provide a base for polymorphic references

More Examples



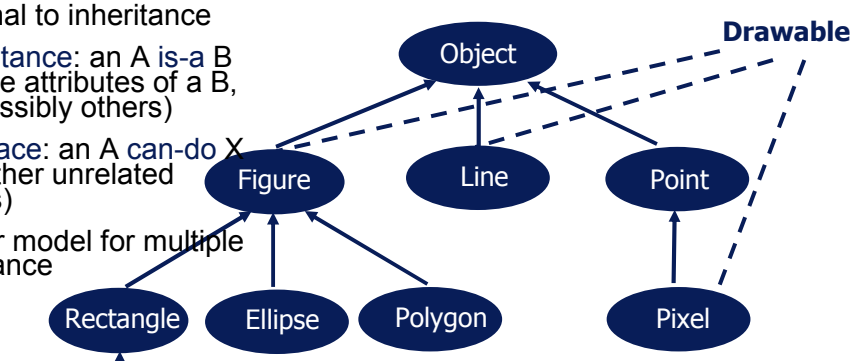
Interfaces

- We've used the term interface to mean the set of service methods provided by an object
- That is, the set of methods that can be invoked through an object define the way the rest of the system interacts, or interfaces, with that object
- The Java language has an **`interface`** construct that formalizes this concept
- Recall the paint-brush application example
  - Suppose we want now to be able to paint also pixels and lines



Interfaces

- Orthogonal to inheritance
  - inheritance: an A is-a B (has the attributes of a B, and possibly others)
  - interface: an A can-do X (and other unrelated actions)
  - better model for multiple inheritance



- More costly at run-time (minor consideration)
- Allow otherwise unrelated classes to satisfy a given requirement
  - We don't have a common view point of these classes that sees the method **`draw`**!

Interfaces

- A Java **interface** is a collection of constants and abstract methods
  - Members are all **public static final** (no need to declare)
- With interfaces we can define a new type of reference with a specific point of view of objects from classes with different superclasses
- We define an interface in a similar way to that of a class
  - But we specify only the instance methods of that class (its interface) without the implementation of these methods
- A class that **implements** an interface must provide implementations for all of the methods defined in the interface
  - This relationship is specified in the header of the class:

```
class class-name implements interface-name {
  }
```


Interfaces and Classes

```
/**
 * An interface for classes whose instances know how to
 * draw themselves on a graphical window
 */
public interface Drawable {
/**
 * Draws this object
 */
public void draw();
}
/**
 * Represents a pixel on a graphical area
 */
public class Pixel extends Point implements Drawable {
    ...}
/**
 * Draws this pixel on the screen
 */
public void draw() {
    ...}
  ...
}
```

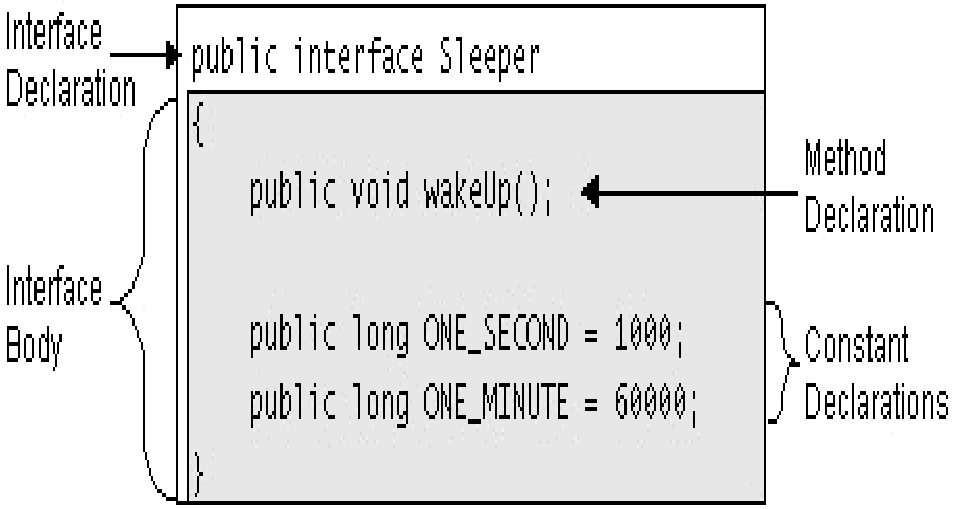
Interfaces

- Unlike interface methods, interface constants require nothing special of the implementing class
  - Constants in an interface can be used in the implementing class as if they were declared locally
  - This feature provides a convenient technique for distributing common constant values among multiple classes
- A method that knows about a particular interface may interact with an object of any of the implementing classes
  - An interface name can be used as a generic reference type name
- A reference to any object of any class that implements that interface is compatible with that type
  - For example, the name of the interface `Drawable`, can be used as the type of a parameter to a method
  - An object of any class, as `Pixel`, that implements `Drawable` can be passed to that method

Interfaces

- An interface can be implemented by multiple classes
  - Each implementing class can provide their own unique version of the method definitions
- A class can be derived from a base class **and** implement one or more interfaces
- An interface can be derived from another interface, using the **`extends`** reserved word
  - The child interface inherits the constants and abstract methods of the parent
- A class that implements the child interface must define all methods in both the parent and child
- An interface is not a class, and cannot be used to instantiate an object
  - Note that the interface hierarchy and the class hierarchy are distinct

Interfaces



Interfaces and Polymorphism

- For polymorphism, an interface plays the role of a superclass: the example of polymorphism works very well if the abstract class `Figure` is an interface
- You can apply polymorphism with classes from different inheritance hierarchies if they implement a same interface
- The advantage of interface is that a class can implement several interfaces
- We can cast (upcast and downcast) between a class and an interface it implements (and upcast between an interface and `Object`)

Interfaces and Abstract Classes

- Note the similarities between interfaces and abstract classes
  - Both define abstract methods that are given definitions by a particular class
  - Both can be used as generic type names for references
- However, a class can implement multiple interfaces, but can only be derived from one class
  - Java does not support multiple inheritance of classes
- A class that implements multiple interfaces specifies all of them in its header, separated by commas
- The ability to implement multiple interfaces provides many of the features of **multiple inheritance**, the ability to derive one class from two or more parents

Interfaces and Abstract Classes

- Recall the paint-brush application example
  - Suppose we want now to be able to paint also pixels and lines

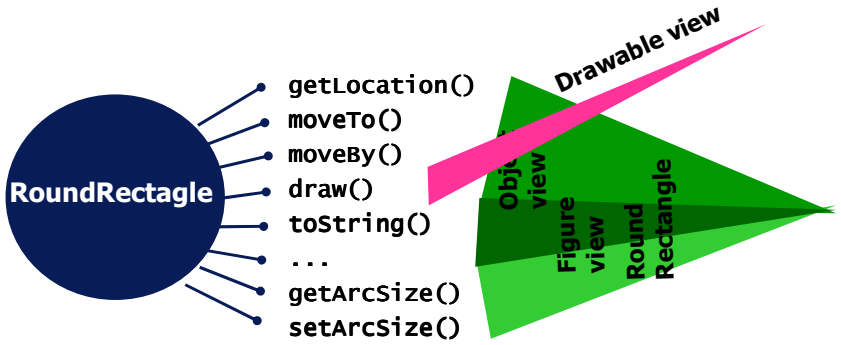
```
/**
 * A geometrical figure. A Figure has a location and size
 * it can draw itself on the screen ...
 */
public abstract class Figure implements Drawable {

    // The top-left corner of the rectangular area that
    // contains the figure
    protected int x,y;
    ...
```

- Here we do not have to declare again the method `draw()`

Interfaces and Abstract Classes

```
/**
 * A picture made of geometrical figures
 */
public class PaintBrushPicture {
// The elements (figures) that compose this picture
// private Vector elements;
    ...
/**
 * Draws this picture
 */
public void draw() {
    for (int i=0; i< elements.size(); i++) {
         ((Drawable)elements.elementAt(i)).draw();
       }
}
```


Interfaces and Abstract Classes



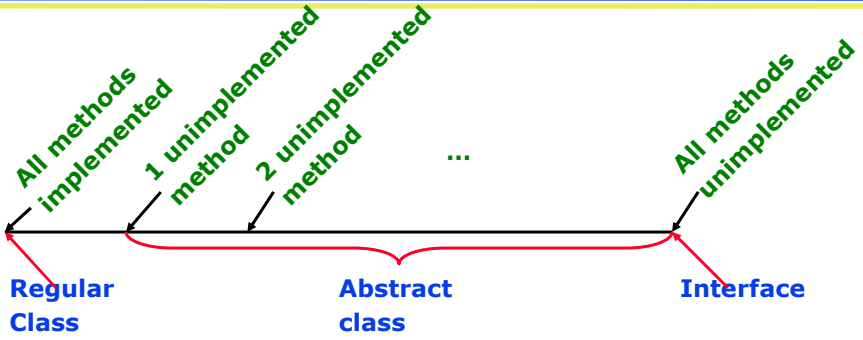
Interfaces and Abstract Classes

```
/**
 * An interface for classes whose instances can move
 */
public interface Moveable {
/**
 * Moves this object to a new location
 * @param x,y The new location for the object
 */
public void moveTo(int x, int y);
/**
 * Moves this object by a given offset
 * @param dx,dy The given offset
 */
public void moveBy(int dx, int dy);
}
```


Interfaces and Abstract classes

```
/**
 * A geometrical figure. A Figure has a location and size
 * it can draw itself on the screen ...
 */
public abstract class Figure implements Drawable, Moveable {
    ...
  /**
   * Moves this figure to a new location
   * @param x,y The coordinates of the new location
   */
  public void moveTo(int x, int y) {
    this.x = x;
    this.y = y;
   }
...
```

Interfaces vs. Abstract classes, How to Decide?



- So why have abstract classes and interfaces?
  - Not all programming languages have Interface's! (e.g., C++)
  - To enforce design conformity
  - Clients can expect a uniform set of methods for related classes
  - Interfaces represent "action ideas", "verbs" or "properties" about class

Empty or Marker Interfaces

- Interface's are used to "tag" a class with a property
- Interface without methods and constants (=> empty interface) can be used to add a property to a class

```
interface Cloneable {}
interface Serializable {}
class X implements Cloneable {}
class X implements Serializable {}`
```

Object Cloning

- All classes inherit a `clone()` method from `Object`
- Returns an identical copy of an object
  - A shallow copy, by default
  - A deep copy is often preferable
- The default `clone` implementation in `Object` is declared as `protected`
  - For a programmer-defined class the `clone()` method is visible only within the class and its subclasses
- The (empty) `Cloneable` interface must be implemented
  - By convention, classes that implement this interface should override the `Object clone()` so that it has `public` scope
  - Invoking `Object`'s `clone()` method on an instance that does not implement the `Cloneable` interface will throw an exception `CloneNotSupportedException`
- Note that this interface does not contain the `clone()` method
  - Therefore, it is not possible to clone an object merely by virtue of the fact that it implements this interface
  - Even if the clone method is invoked reflectively, there is no guarantee that it will succeed

The Default clone Method Implementation

```
class Test {
  private int n;
  public Test (int i) {n = i;}
}
public class CloneTest {
   public static main (String[] args) {
   Test t1 = new Test(1);
   Test t2 = (Test) t1.clone(); // ERROR !!!
   }
}
```

The protected clone method that Test inherit from Object is not visible in class CloneTest

```
class Test implements Cloneable {
private int n;
public Object clone() throws CloneNotSupportedException{
       return super.clone();}
}
```

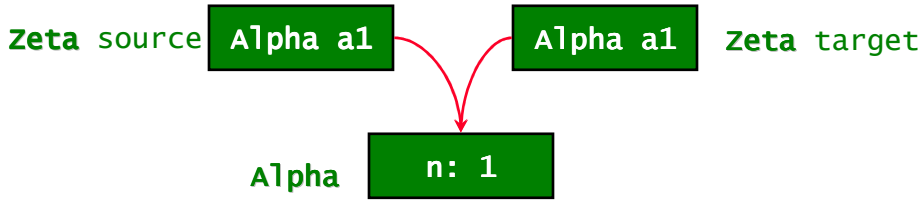
Flexibility in Overriding Methods

- A method can only override a method in its superclass if the superclass method is not private
- There is some flexibility in the visibility of an overriding method, whether the arguments are final or not, and the exceptions thrown by it
  - An overriding method may be more visible than that in the superclass
  - Arguments that are final in the superclass version do not have to be marked as such in the subclass version
  - Any checked exception thrown by the subclass version must match the type of the one thrown by the superclass version, or be a subclass of such an exception
    - However, the subclass version does not have to throw any exceptions that are thrown by the superclass version
- One reason for the last rule is that the overriding version of a method is actually able to invoke the superclass version of the same method
  - In doing so, it might choose to catch and handle the exceptions for itself, rather than propagating them

Problems with the default `clone` Method

```
class Alpha implements Cloneable {
int n;
public void set (int i) {n = i;}
public void get() {return n;}
public Object clone() throws CloneNotSupportedException{
       return super.clone();}
}

class Zeta implements Cloneable {
Alpha al = new Alpha();
public void set (int i) {al.set(i);}
public void get() {return al.get();}
public Object clone() throws CloneNotSupportedException{
       return super.clone();} // Caution !!!
}
```

Problems with the default clone Method




```
class CloneProblem {
  public static main (String[] args) throws
                          CloneNotSupportedException{
Zeta z1 = new Zeta();         //Creates one Zeta object
Z1.set(1);                    //z1.alpha == 1
Zeta z2 = (Zeta) z1.clone(); //clone the object
Z2.set(2);                    //z2.alpha == 2
System.out.println(z1.get());//Not well: z1.alpha == 2
```


A Correct Object Cloning Solution

```
class Alpha implements Cloneable {
int n;
public void set (int i) {n = i;}
public int get() {return n;}
public Object clone() throws CloneNotSupportedException{
   return super.clone();}
}
class Zeta implements Cloneable {
Alpha al = new Alpha();
public void set (int i) {al.set(i);}
public int get() {return al.get();}
public Object clone() throws CloneNotSupportedException{
   Zeta t = (Zeta)super.clone();//1st: Use the sc clone
   t.al = (Alpha) al.clone();//2nd:Invoke Alpha's clone
   return t;                  //3rd:Return the exact copy
}
```

Disabling Object Cloning

- Disabling object cloning might be necessary
  - A unique attribute - database lock or open file reference
- The default `clone()` restricts cloning because of the method's protected scope
  - An inherited member (e.g., protected) cannot be redeclared as private
  - Cloning cannot be disabled by redeclaring the default `clone()` as private
- Not sufficient to omit `Cloneable`
  - Sub classes might implement it

```
class NotCloneable {
    ...
    // Explicitly disabling subclass cloning
    public Object clone()
                    throws CloneNotSupportedException {
        throw new CloneNotSupportedException("...");}
    ...
}
```


Comparing Objects

- A number of programs perform comparisons between various elements…
  - These elements can be numbers, strings, pairs, etc. In the spirit of generic programming and code reuse we would like the objects manipulated by these programs to be as general as possible
  - But just using the type **Object** is not enough… it does not make sense to compare two arbitrary objects
- There are two solutions to the problem (two design patterns)
  - One is to encapsulate the *knowledge* about which objects are comparable and how to compare them in an… object! Such an object implements the `Comparator` interface (`compare()` method) and is used for designing a custom ordering scheme
  - The other is to make the objects we compare implement themselves a comparison operation via the `Comparable` interface (`compareTo()`) and it used for designing a natural ordering schema

The Comparator interface

```
  int compare(Object o1, Object o2);
   // compare(o1,o2) is
   // negative if o1 is strictly less than o2
   // zero if o1 is equal to o2
   // positive if o1 strictly greater than o2
   // throws ClassCastException if the arguments'
   // types prevent them from being compared
} // End of interface Comparator
```

- A class's custom ordering is said to be consistent with equals if and only if `(compare((Object)e2, (Object)e1)==0)` has the same boolean value as:
  `e1.equals((Object)e2)` for every `e1` and `e2` of class C

The Comparable interface

```
public interface Comparable {
  int compareTo(Object o);
   // this.compareTo(o) is
   // negative if this is strictly less than o
   // zero if this is equal to o
   // positive if this strictly greater than o
   // throws ClassCastException if the type of o
   // prevents it from being compared to this
}
```

- A class's natural ordering is said to be consistent with equals if and only if `(e1.compareTo((Object)e2)==0)` has the same boolean value as:
  `e1.equals((Object)e2)` for every `e1` and `e2` of class C
- This interface is already implemented by existing classes such as `Character`, `Double`, `Float`, `Long`, `String`, `Integer` and others

Example of Comparator: Integer Ordering

```
import java.util.*;

  class IntComparator implements Comparator {
    public int compare(Object obj1, Object obj2) {
      int i1 = ((Integer)obj1).intValue();
      int i2 = ((Integer)obj2).intValue();

      return Math.abs(i1) - Math.abs(i2);
    }
 }
```


Example of Comparator: Integer Ordering

```
class collect {
 public static void main(String args[]) {
    Vector vec = new Vector();
    vec.addElement(new Integer(-200));
    vec.addElement(new Integer(100));
    vec.addElement(new Integer(400));
    vec.addElement(new Integer(-300));
    Collections.sort(vec);
    for (int i = 0; i < vec.size(); i++) {
      int e=((Integer)vec.elementAt(i)).intValue();
      System.out.println(e);
    }
    System.out.println("===========");
    Collections.sort(vec, new IntComparator());
    for (int i = 0; i < vec.size(); i++) {
      int e=((Integer)vec.elementAt(i)).intValue();
      System.out.println(e);
     }
 }
}
```

-300
-200
100
400

100
-200
-300
400

Example of Comparable: Lexicographic Ordering

```
import java.util.*;
public class Name implements Comparable {
    private String first;
    private String last;
    public Name(String firstName, String lastName ) {
      first = firstName;
      last = lastName; }
    public String getFirst() {
      return first; }
    public String getLast() {
      return last; }
    public String toString() {
      return first + " " + last; }
    public int hashCode() {
      return first.hashCode() + last.hashCode(); }
```


Example of Comparable: Lexicographic Ordering

```
    public boolean equals( Object o ) {
      boolean retval = false;
      if (o !=null && o instanceof Name ) {
          Name n = (Name)o;
          retval = n.getFirst().equals( first ) &&
                   n.getLast().equals( last ); }
      return retval;
    }
    public int compareTo( Object o ) throws
                          ClassCastException {
      int retval;
      Name n = ( Name ) o;
      retval = last.compareTo( n.getLast() );
      if ( retval == 0 )
          retval = first.compareTo( n.getFirst() );
      return retval;
    }
} //Name
```

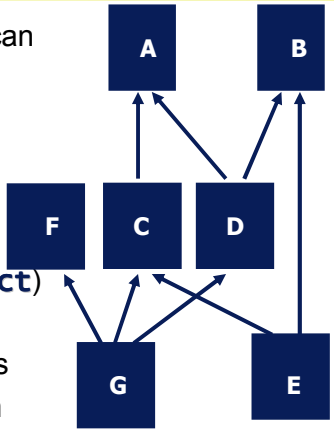
Example of Comparable: Lexicographic Ordering

```
class   classNameTest {
    public static void main (String[] a){
    Name n1 = new Name("Giorgos","Tziritas");
    Name n2 = new Name("Giorgos","Georgakopoulos");
    Name n3 = new Name("Aggelos","Bilas");
    Vector v = new Vector();
    v.add(n1);
    v.add(n2);
    v.add(n3);
    Collections.sort(v);
    System.out.println(v);
    }
}
```

[Aggelos Bilas, Giorgos Georgakopoulos, Giorgos Tziritas

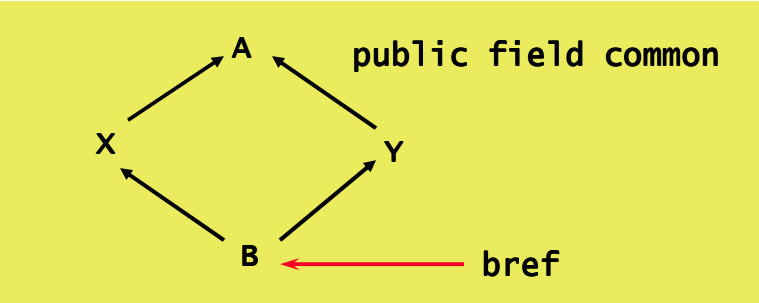
Multiple Inheritance

- Multiple inheritance means that a derived class can have more than one parent
- Some programming languages allow for multiple inheritance (e.g., C++)
- In Java:
  - We can have Multiple Interface Inheritance
  - Single Class Inheritance (implicitly class **`Object`**)
- Types of possible inheritance:
  - An interface can inherit from multiple interfaces
  - A class (abstract or otherwise) can inherit from multiple interfaces by implementing
  - A class (abstract or otherwise) can inherit from a **single** other (abstract or normal) class by extending



Single Inheritance vs. Multiple Inheritance

- name conflict (diamond inheritance)

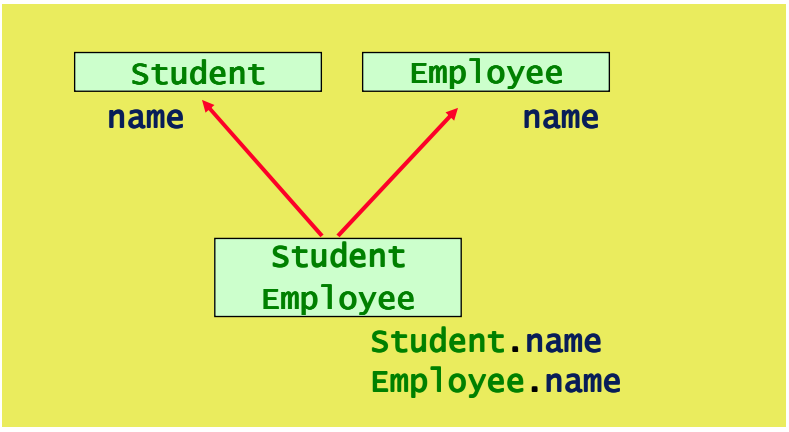


`bref.common = X or Y or A's common field?`

- Multiple inheritance leads to many potentially confusing naming problems
  - Growing consensus: the benefits of multiple inheritance *aren't worth the problems*

Multiple Inheritance and Conflict Resolution

- Case of name conflict of fields
  - if two supertypes have fields of the same name
    - then you must use the qualified name



Multiple Inheritance and Conflict Resolution

- Case of ***name-conflict*** of methods
  - if methods in supertypes have same name but different numbers or types of parameters
    - then subclass' interface will have two overloaded methods with the same name but different signatures
    - such a method will have multiple bodies (one for each signature)
  - if methods in supertypes have exactly the same signature
    - then subclass' interface will have one method with that signature (i.e., a single body)
  - if methods in supertypes differ only in return type
    - cannot implement both interfaces
  - if methods in supertypes differ only in the types of exceptions they throw
    - there must be only one implementation of subtype that satisfies both supertypes' throws clauses (i.e., common subset of exceptions)
    - alternatively, the combined method can throw no exceptions

Try yourselves

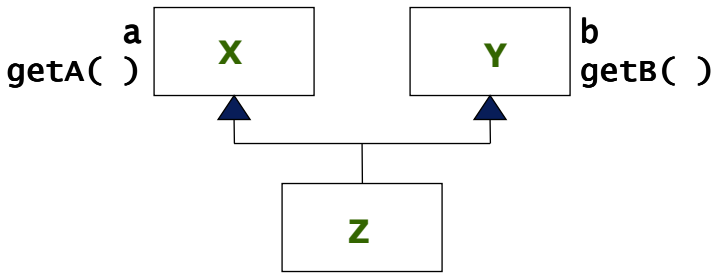
```
abstract class AA {
   public void methodX() { System.out.println("BOOM");}
}
interface  IA {
  final static String consStr="Const of IA";
  void methodX();
}
interface IB {
  final static String consStr="Const of IB";
  void methodX();
}
interface IC extends IA, IB {};
 // an interface can extend >1 other interfaces
```

Try yourselves

```
class RealClass extends AA implements IA, IB {}
      // the obligation to implement methodX is satisfied
      // because an implementation is inherited from AA

class tester {
  public static void main (String[] a){
      RealClass rc = new RealClass();
      rc.methodX();
      //System.out.println(rc.consStr);  // The compiler will
                  // identify that the above line is ambiguous

      System.out.println(((IA)rc).consStr);
      System.out.println(((IB)rc).consStr);
  }
}
```

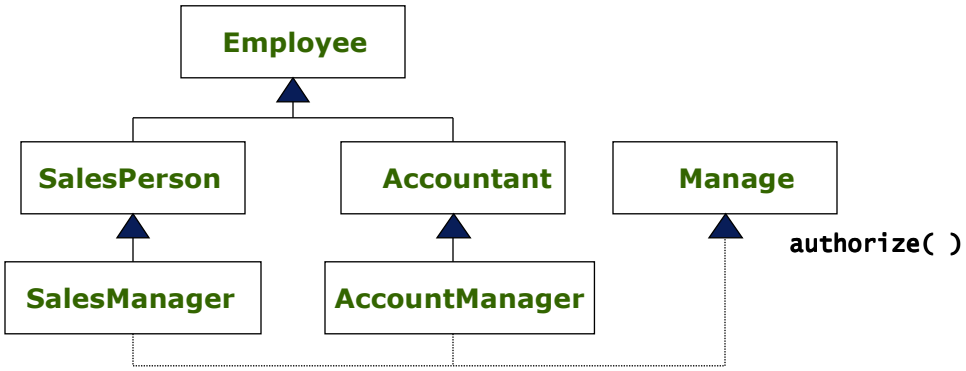

Limitations of Interface for Multiple Inheritance

- An interface does not provide a natural means of realizing multiple inheritance in situations where there is no inheritance conflicts



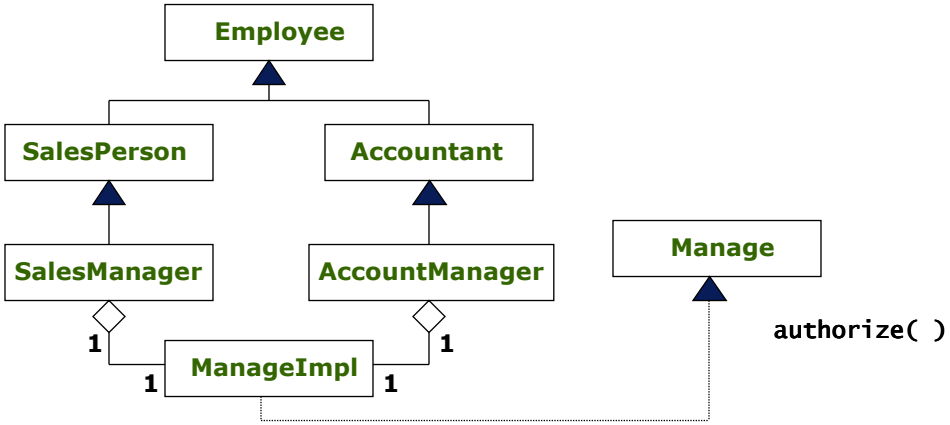
Limitations of Interface for Multiple Inheritance

- While the principle for inheritance is code reusability, the interface facility does not encourage code reuse because of the duplication of the implementation



Limitations of Interface for Multiple Inheritance

- Duplication of implementation sometimes can be avoided by the technique of method forwarding



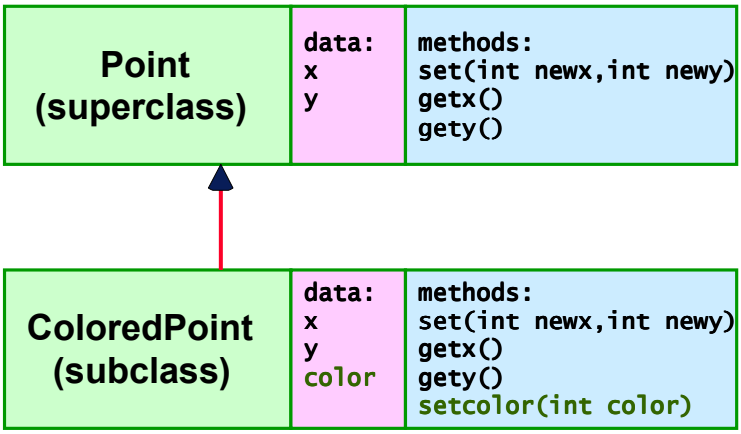
How Class C2 can Reuse Code of a Class C1?

- Inherits from C1
  - Use inheritance to represent the relation *is-a* between classes
  - Don't use inheritance just to reuse code
- Composition:
  - an instance variable of the C2 is of type C1
- Delegation:
  - C2 delegates the execution of one of its methods to an instance of C1;
  - this instance can be an instance variable of C2 created by the method of C2 to do the work

Forms of Inheritance

- Specialization
  - New class is a *specialized variety* of the parent class
  - Always creates a subtype
- The most ideal, and most common form of inheritance
  - Good design should strive for this kind of inheritance

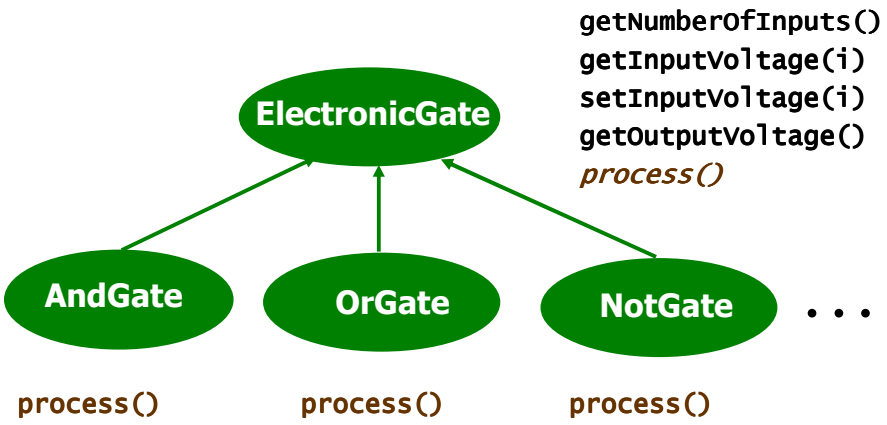
Specialization



Forms of Inheritance

- Specification
  - The parent class *specifies* some behavior but *does not implement* the behavior
  - The child class implements the methods described
- A form of specialization
  - not refinements, but actually realizations of incomplete, abstract specs
- Two ways of implementation:
  - Interface
  - Abstract class
    - At least one abstract method => class abstract
    - Abstract class => no direct instances/objects

Specification



Forms of Inheritance

- Construction
  - Sometimes a class can inherit almost all desired functionality without any conceptual/subtype relationship (Pragmatic)
    - The child class uses the *parent class' constructor* [e.g. class Hole extends Ball …]
    - The **`super()`** keyword is used
- May be used when parent and child fail to share any abstract relationship
  - Sometimes frowned upon, since it often directly breaks the principle of substitutability (forming subclass that are not subtypes)
  - On the other hand, because it is often a fast and easy route to developing new data abstractions, it is nevertheless widely used

Construction

```
class Stack extends Vector {
  public Stack()
      {super();}
  public Object push(Object item)
      {addElement(item); return item;}
  public boolean empty
      {return isEmpty();}
  public synchronized Object pop()
      {Object obj = peek();
       removeElementAt(size() -1);
       return obj;}
  public synchronized Object peek()
      {return ElementAt(size() -1);}
}
```


Forms of Inheritance

- Extension
  - Child class *only adds* new behavior to parent class
  - Does not modify or alter any inherited members
- As the functionality of the parent remains available and untouched, subclassification for extension does not contraverce the principle of substitutability
  - So subclasses are always subtypes

Extension

```
class Properties extends HashTable {
  public synchronized void load(InputStream in) throws
    IOException {...}
  public synchronized void save(OutputStream out,
                                String header)
    {...}
  public String getProperty(String Key)
    {...}
  public Enumeration propertynames()
    {...}
  public void list(PrintStream out)
    {...}
}
```


Forms of Inheritance

- Limitation
  - The child class is more *restrictive* than the behavior of the parent class (usually when parent can/should not be changed)
  - Used when building on a set of base classes that should not be modified
- Take a parent method and make it illegal
  - Override parent class methods, essentially making them useless
- Such subclasses are not subtypes
  - Thus, should be avoided

68

Limitation

```
class Set extends Vector {
  // methods addElement, removeElement, contains,
  // isEmpty and size are all inherited from Vector

  public int indexof (Object obj)
   {System.out.println("Do not use Set.indexof");
    return 0;}

  public Object elementAt (int index)
   {return null;}
}
```

- Ideally, would use exceptions, but illegal (would change type-signature)
- Actually, not possible as indexOf and elementAt are final in class Vector

Forms of Inheritance

- Combination
  - Child class *combines* two or more abstractions
    - multiple inheritance (teaching assistant is both a teacher and a student)
  - Not directly supported in Java
    - Implement more than one interface
- Usually subtype of all parts

Summary of Inheritance

- *Specialization*: The child class is a special case of the parent class; in other words, the child class is a subtype of the parent class
- *Specification*: The parent class defines behavior that is implemented in the child class but not in the parent class
- *Construction*: The child class makes use of the behavior provided by the parent class, but is not a subtype of the parent class
- *Extension*: The child class adds new functionality to the parent class, but does not change any inherited behavior
- *Limitation*: The child class restricts the use of some of the behavior inherited from the parent class
- *Combination*: The child class inherits behavior from two or more parent classes

Drawbacks with Inheritance

- Static: if an object must play different roles at different moments, it is not possible to use inheritance
  - For instance, if a person can be a pilot or a client of an airplane company, 3 classes **`Person`**, with 2 subclasses **`Client`** and **`Pilot`** are no use
- Poor encapsulation for the superclass
  - Often difficult to change the superclass, especially if protected variables have been used
- Not possible to inherit from a final class (**`String`** for instance)