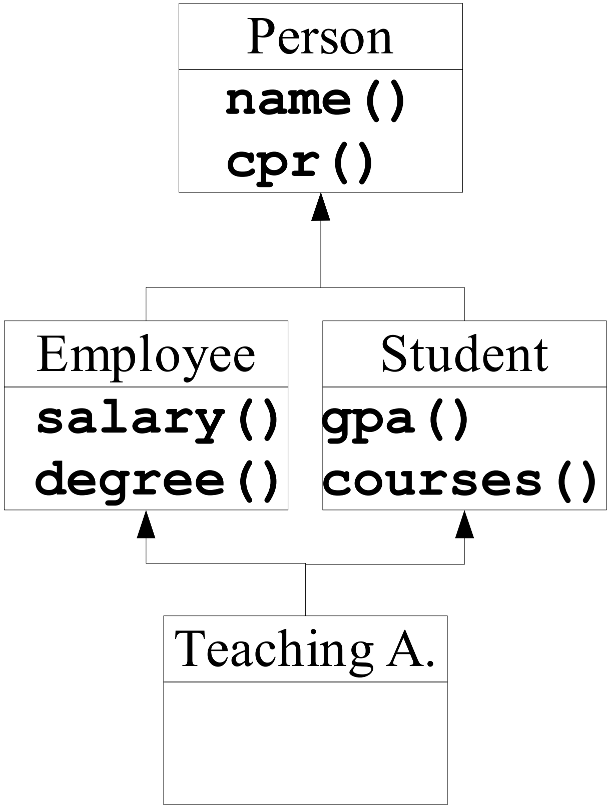
# The Interface Concept

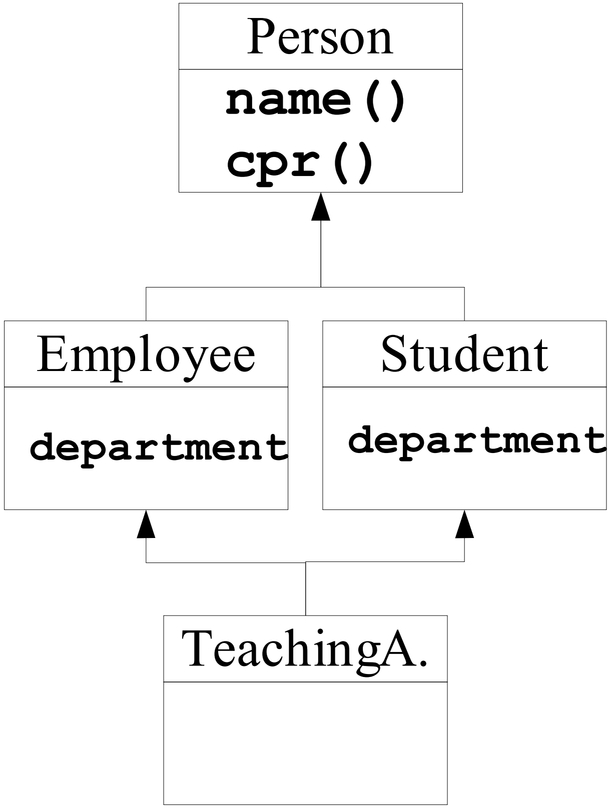
- Multiple inheritance
- Interfaces
- Four often used Java interfaces
  - Iterator
  - Cloneable
  - Serializable
  - Comparable

# Multiple Inheritance, Example



- For the teaching assistant when want the properties from both Employee and Student.

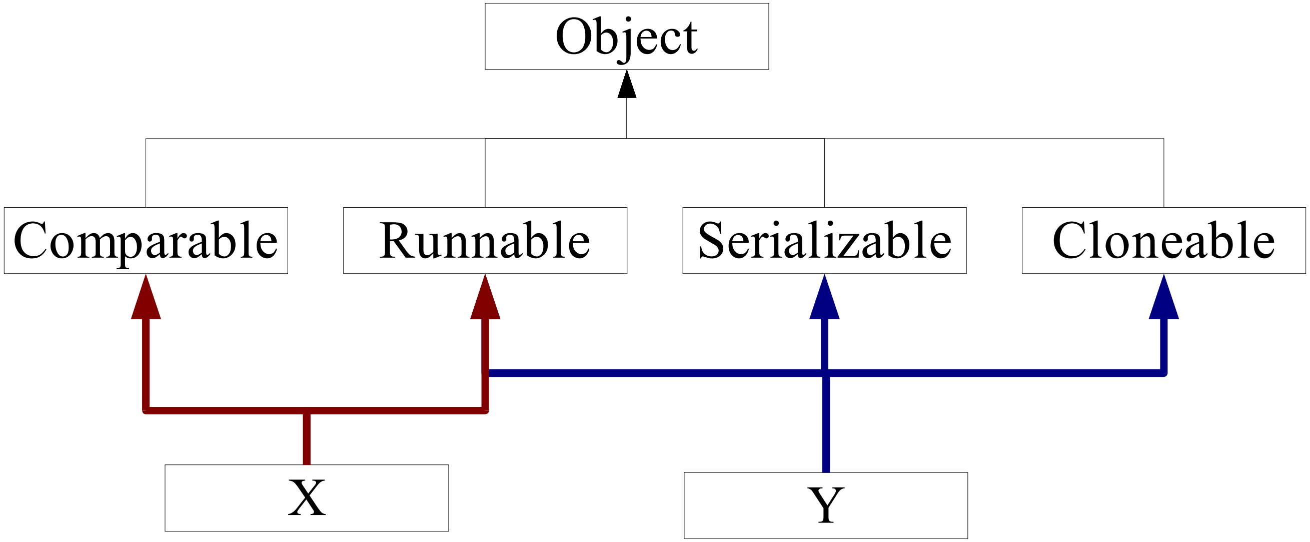
# Problems with Multiple Inheritance



```
ta = new TeachingAssistant();
ta.department;
```

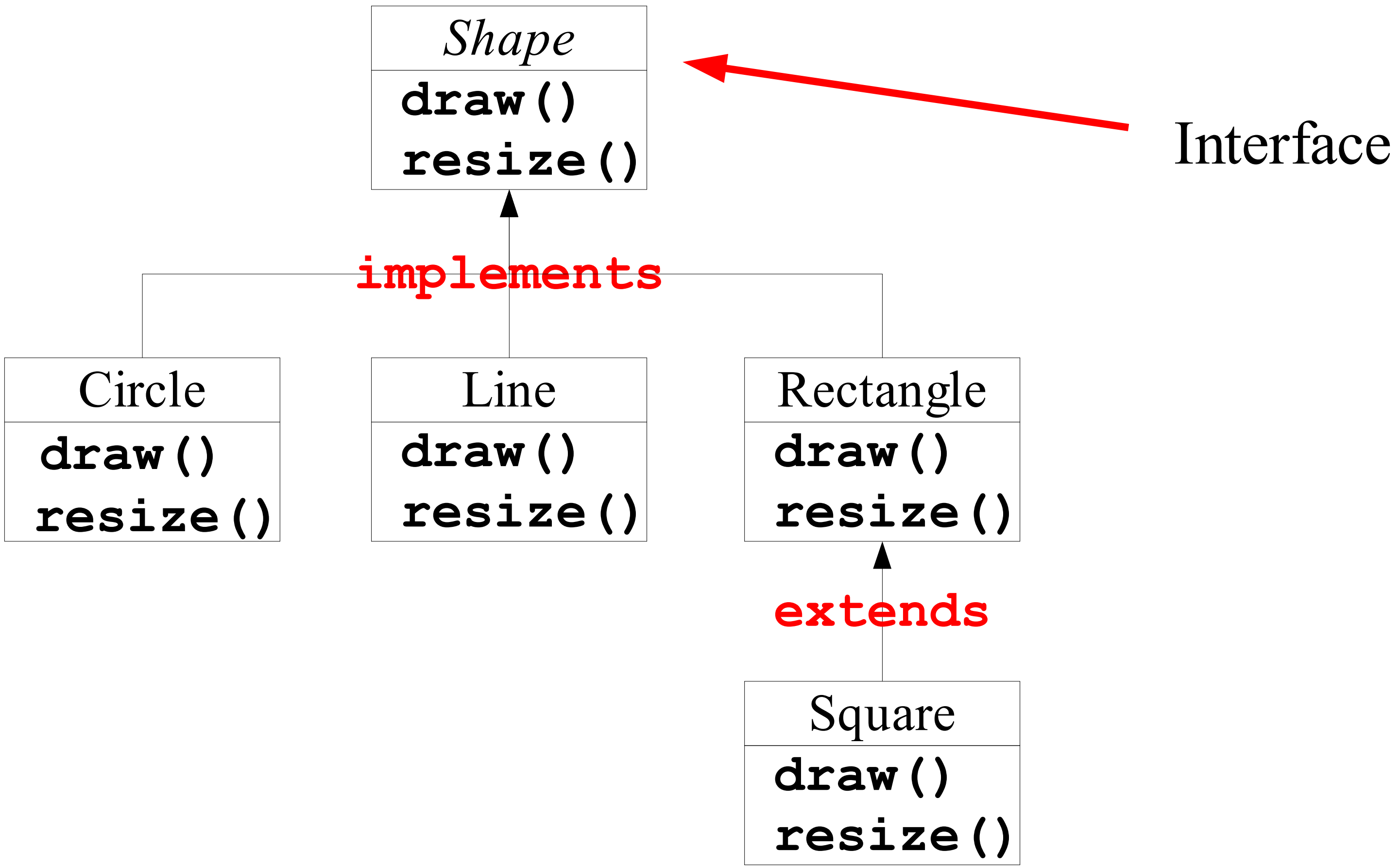
- Name clash problem: Which **department** does **ta** refers to?
- Combination problem: Can **department** from Employee and Student be combined in Teaching Assistant?
- Selection problem: Can you select between **department** from Employee and **department** from Student?
- Replication problem: Should there be two **departments** in TeachingAssistent?

# Multiple Classifications



- Multiple and overlapping classification for the classes X and Y, i.e.,
  - class X is Runnable and Comparable
  - class Y is Runnable, Serializable, and Cloneable

# Java's **`interface`** Concept



# Java's `interface` Concept, cont.

```
public interface Shape {
    double PI = 3.14;   // static and final => upper case
    void draw();        // automatic public
    void resize();      // automatic public
}

public class Rectangle implements Shape {
    public void draw() {System.out.println ("Rectangle"); }
    public void resize() { /* do stuff */ }

}

public class Square extends Rectangle {
    public void draw() {System.out.println ("Square"); }
    public void resize() { /* do stuff */ }

}
```

# Java's `interface` Concept

- An *interface* is a collection of method declarations.
  - An interface is a class-like concept.
  - An interface has no variable declarations or method bodies.

- Describes a set of methods that a class can be forced to implement.
- An interface can be used to define a set of “constant”.
- An interface can be used as a type concept.
  - Variable and parameter can be of interface types.
- Interfaces can be used to implement multiple inheritance like hierarchies.

# Java's **interface** Concept, cont.

```
interface InterfaceName {
   // "constant" declarations
   // method declarations
}

// inheritance between interfaces
interface InterfaceName extends InterfaceName {
   ...
}

// not possible
interface InterfaceName extends InterfaceName1, InterfaceName2
{
   ...
}

// not possible
interface InterfaceName extends ClassName {  ... }
```

# Java's **interface** Concept, cont.

```
// implements instead of extends
class ClassName implements InterfaceName {
   ...
}

// multiple inheritance like
class ClassName implements InterfaceName1, InterfaceName2
{
   ...
}

// combine inheritance and interface implementation
class ClassName extends SuperClass implements InterfaceName
{
   ...
}

// multiple inheritance like again
class ClassName extends SuperClass
         implements InterfaceName1, InterfaceName2 {
   ...
}
```

# Semantic Rules for Interfaces

- Type
  - An interface can be used as a type, like classes
  - A variable or parameter declared of an interface type is polymorph
    - Any object of a class that implements the interface can be referred by the variable
- Instantiation
  - Does not make sense on an interface.
- Access modifiers
  - An interface can be public or “friendly” (the default).
  - All methods in an interface are default abstract and public.
    - Static, final, private, and protected cannot be used.
  - All variables (“constants”) are public static final by default
    - Private, protected cannot be used.

# Some of Java's Most used Interfaces

- **`Iterator`**
  - To run through a collection of objects without knowing how the objects are stored, e.g., in array, list, bag, or set.
  - More on this in the lecture on the Java collection library.
- **`Cloneable`**
  - To make a copy of an existing object via the **`clone()`** method on the class **`Object`**.
  - More on this topic in todays lecture.
- **`Serializable`**
  - Pack a web of objects such that it can be send over a network or stored to disk. An naturally later be restored as a web of objects.
  - More on this in the lecture on Java's I/O system
- **`Comparable`**
  - To make a total order on objects, e.g., 3, 56, 67, 879, 3422, 34234
  - More on this topic in todays lecture.

# The `Iterator` Interface

- The **`Iterator`** interface in the package **`java.util`** is a basic iterator that works on collections.

```
package java.util;
public interface Iterator {
    // the full meaning is public abstract boolean hasNext()
    boolean hasNext();
    Object next();
    void remove(); // optional throws exception
}


// use an iterator
myShapes = getSomeCollectionOfShapes();
Iterator iter = myShapes.iterator();
while (iter.hasNext()) {
  Shape s = (Shape)iter.next(); // downcast
  s.draw();
}
```

# The **`Cloneable`** Interface

- A class X that implements the **`Cloneable`** interface tells clients that X objects can be cloned.
- The interface is empty, i.e., has no methods.
- Returns an identical copy of an object.
  - A *shallow copy*, by default.
  - A *deep copy* is often preferable.

- Prevention of cloning
  - Necessary if unique attribute, e.g., database lock or open file reference.
  - Not sufficient to omit to implement **`Cloneable`**.
    - Subclasses might implement it.
  - **`clone`** method should throw an exception:
    - **`CloneNotSupportedException`**

# The **Cloneable** Interface, Example

```
// Car example revisited
public class Car implements Cloneable{
    private String make;
    private String model;
    private double price;
   // default constructor
   public Car() {
        this("", "", 0.0);
    }
   // give reasonable values to instance variables
   public Car(String make, String model, double price){
        this.make  = make;
        this.model = model;
        this.price = price;
    }
    // the Cloneable interface
    public Object clone(){
        return new Car(this.make, this.model, this.price);
    }
}
```

# The **Serializable** Interface

- A class X that implements the **Serializable** interface tells clients that X objects can be stored on file or other persistent media.
- The interface is empty, i.e., has no methods.

```
public class Car implements Serializable {
  // rest of class unaltered
  snip
}

// write to and read from disk
import java.io.*;
public class SerializeDemo{
    Car myToyota, anotherToyota;
    myToyota = new Car("Toyota", "Carina", 42312);
    ObjectOutputStream out = getOutput();
    out.writeObject(myToyota);

    ObjectInputStream in = getInput();
    anotherToyota = (Car)in.readObject();
}
```

# The **Comparable** Interface

- In the package **java.lang**.
- Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

```
package java.lang;
public interface Comparable {
    int compareTo(Object o);
}
```

# The Comparable Interface, Example

```
// IPAddress example revisited
public class IPAddress implements Comparable{
    private int[] n; // here IP stored, e.g., 125.255.231.123

    /** The Comparable interface */
    public int compareTo(Object o){
        IPAddress other = (IPAddress) o; // downcast
        int result = 0;
        for(int i = 0; i < n.length; i++){
            if (this.getNum(i) < other.getNum(i)){
                result = -1;
                break;
            }
            if (this.getNum(i) > other.getNum(i)){
                result = 1;
                break;
            }
        }
        return result;
    }
}
```

# Interface vs. Abstract Class

| Interface | Abstract Class |
|---|---|
| • Methods can be declared | • Methods can be declared |
| • No method bodies | • Method bodies can be defined |
| • “Constants” can be declared | • All types of variables can be declared |
| • Has no constructors | • Can have constructors |
| • Multiple inheritance possible | • Multiple inheritance not possible |
| • Has no top interface | • Always inherits from **`Object`** |
| • Multiple “parent” interfaces | • Only one “parent” class |

# Interfaces and Classes Combined

- By using interfaces objects do not reveal which classes the belong to.
  - With an interface it is possible to send a message to an object without knowing which class(es) it belongs. The client only know that certain methods are accessible.
  - By implementing multiple interfaces it is possible for an object to change role during its life span.

- Design guidelines
  - Use classes for specialization and generalization
  - Use interfaces to add properties to classes.

# Multiple Inheritance vs. Interface

| Multiple Inheritance | Interface |
|---|---|
| • Declaration and definition is inherited. | • Only declaration is inherited. |
| • Little coding to implement subclass. | • Must coding to implement an interface. |
| • Hard conflict can exist. | • No hard conflicts. |
| • Very hard to understand (C++ close to impossible). | • Fairly easy to understand. |
| • Flexible | • Very flexible. Interface totally separated from implementation. |

# Summary

- Purpose: Interfaces and abstract classes can be used for program design, not just program implementation [Meyer pp 239 ff].
- Java only supports single inheritance.
- Java “fakes” multiple inheritance via interfaces.
  - Very flexible because the object interface is totally separated from the objects implementation.

# The **Cloneable** Interface, Example 2

```
package geometric; // [Source: java.sun.com]

/** A cloneable Point */
public class Point extends java.awt.Point implements Cloneable
{
    // the Cloneable interface
    public Object clone(){
        try {
            return (super.clone()); // protected in Object
        }
        // must catch exception will be covered later
        catch (CloneNotSupportedException e){
            return null;
        }
  }
  public Point(int x, int y){
    super(x,y);
  }
}
```