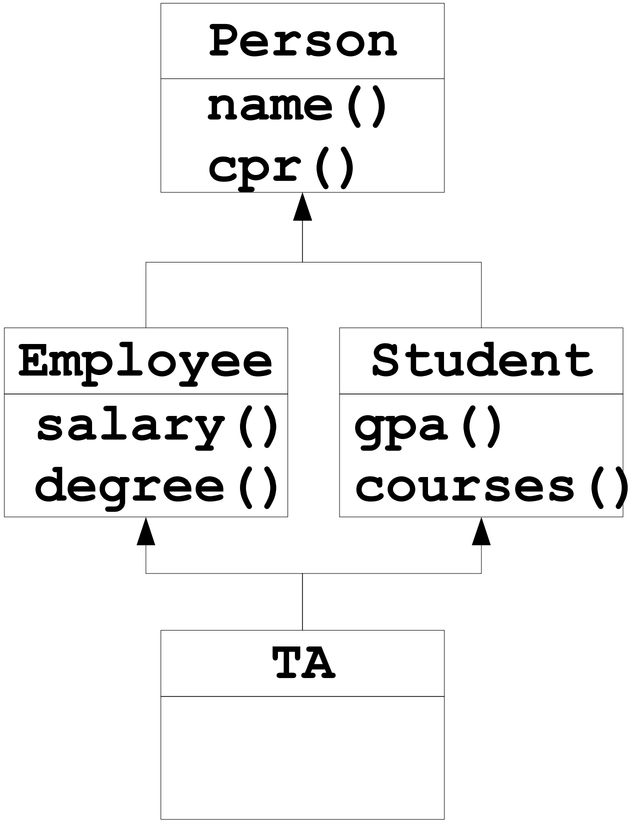
The Interface Concept

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

- Complete story available after lecture on generics!!!

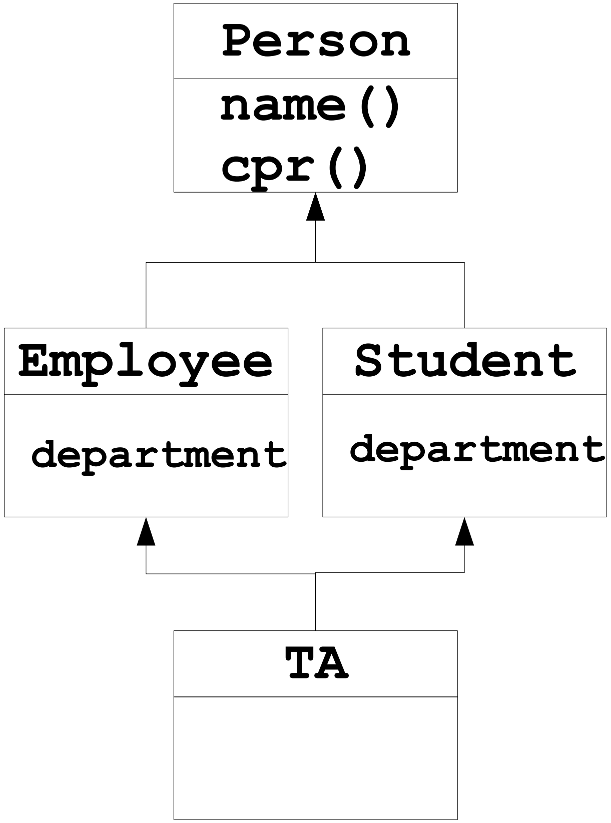
Multiple Inheritance, Example



- For the teaching assistant (**`TA`**) we want the properties from both **`Employee`** and **`Student`**.

- Is this a great idea?

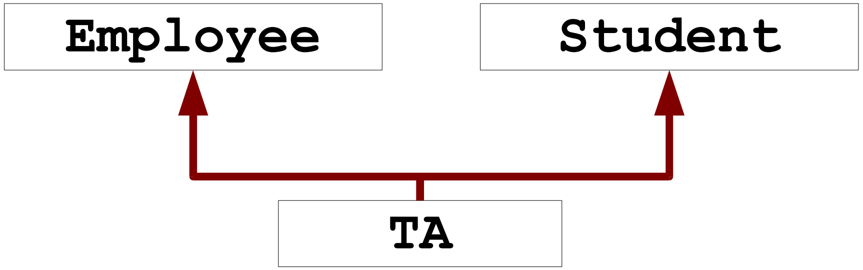
Some Problems with Multiple Inheritance



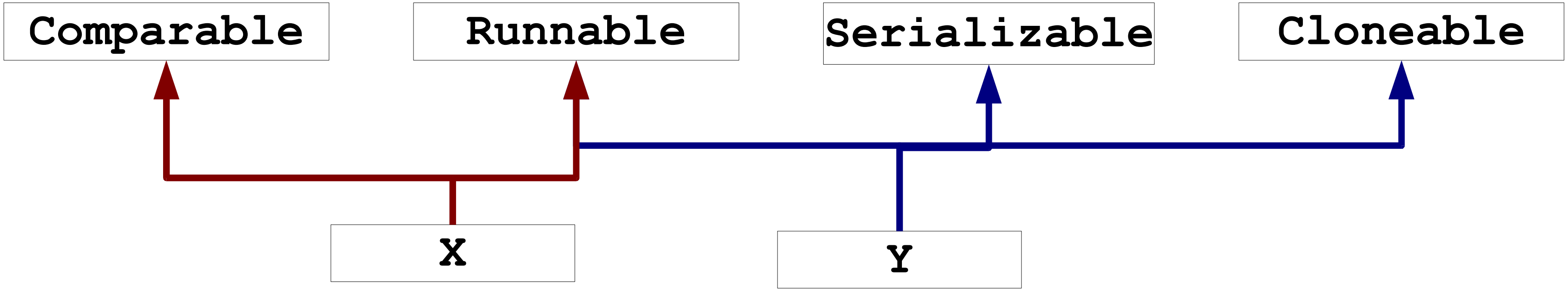
```
ta = new TA();
ta.department = “CS”;
```

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

Multiple Classifications



- Multiple classification for the class **TA**.
  - It is “employee-able” and “student-able”



- Multiple and overlapping classification for the classes **X** and **Y**
  - Class **X** is **Runnable** and **Comparable**.
  - Class **Y** is **Runnable**, **Serializable**, and **Cloneable**.

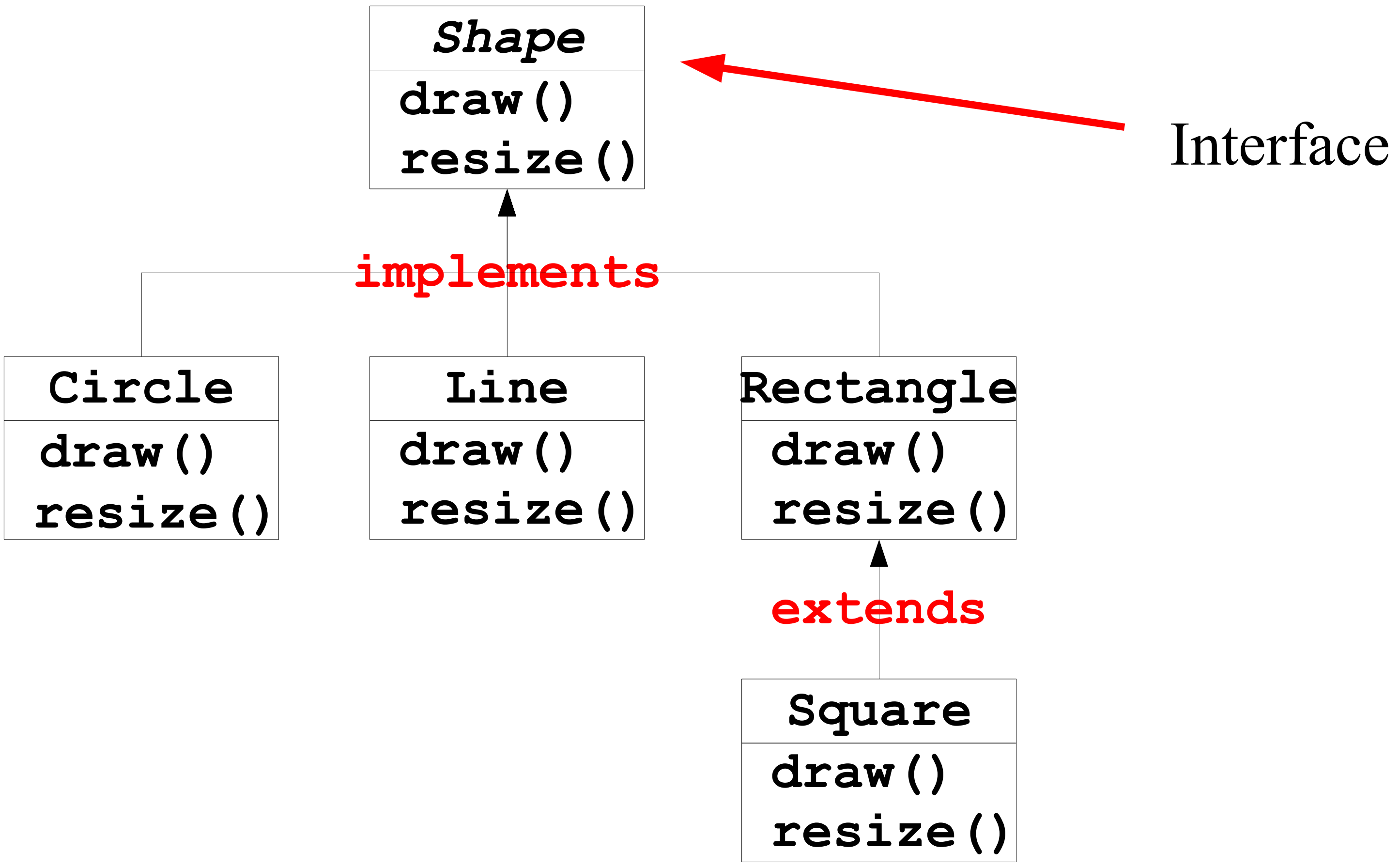
Java's **interface** Concept

```
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, cont



Java's `interface` Concept, cont.

```java
public class UseShape{
    /** Use the Shape interface as a parameter */
    public static void shapeAsParameter(Shape sh){
        sh.draw();
    }

    /** Use the Shape interface as a return type */
    public static Shape getAShape(){
        return new Line();
    }



    public static void main(String[] args){
        /** Use the Shape interface as a type */
        Shape s1 = new Circle();
        Shape s2 = getAShape();
        shapeAsParameter(s1);
        s2.draw();
    }
}
```

Java's `interface` Concept

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

- Describes a set of methods that a class can be forced to implement.

- Can be used to define a set of “constants”.
- Can be used as a type concept.
- Can be used to implement multiple inheritance like hierarchies.

Combining Multiple **interface**s

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

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

// extends multiple interfaces (multiple inheritance like)
interface InterfaceName extends InterfaceName1, InterfaceName2
{
   ...
}

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

Interfaces and Classes Combined

```
// 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 {
   ...
}

// not possible!
class ClassName extends InterfaceName {...}
```

Interfaces and Classes Combined, cont.

- By using interfaces objects do not reveal which classes the belong to.
  - It is possible to send a message to an object without knowing which class(es) it belongs.
  - 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

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.

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

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.

What is Ugly/Wrong?

```
public interface A1 {
   int getA();
}
public interface A2 {
   double getA();
}
public interface A3 extends A1, A2{
   //more stuff
}
```

```
public interface B1 {
   int getA();
}
public interface B2 extends B1 {
   int getB();
}
public interface B3 extends B1, B2{
   //more stuff
}
```

What is Ugly/Wrong, cont. ?

```
public interface C1 extends C3{
   int getA();
}
public interface C2 extends C1 {
   int getB()
}
public interface C3 extends C1{
  int getC()
}
```

```
public interface D1 {
   int getA()
}
public class DoesD1 implements D1{
   int getA() {return 42;}
}
```

```
public interface E1 {
}
```

Some of Java's most used Interfaces

- **`Iterator`**
  - Runs through a collection of objects in an array, list, bag, or set.
  - More on this in the lecture on the Java collection library.
- **`Cloneable`**
  - Copies an existing object via the **`clone()`** method
  - More on this topic in todays lecture.
- **`Serializable`**
  - Packs or ships a web of objects (file or network).
  - More on this in the lecture on Java's I/O system
- **`Comparable`**
  - Makes a total order on objects, e.g., 3, 56, 67, 879, 3422, 34234
  - More on this topic in todays lecture.

Stuff you often need to do in a software project!!

The **Iterator** Interface

- **java.util.Iterator** is a basic iterator that works on all 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 has no methods
  - An “empty” interface
- 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 {
    // instance variables
    private String make;
    private String model;
    private double price;
   // give reasonable values to instance variables
   public Car(String make, String model, double price){
        this.make  = make;
        this.model = model;
        this.price = price;
    }
    // the clone method 1.4
    public Object clone(){
        return new Car(this.make, this.model, this.price);
    }
    // the clone method 5.0
    public Car clone(){
        return new Car(this.make, this.model, this.price);
    }
}
```

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
        }
        catch (CloneNotSupportedException e){
            return null;
        }
  }
  public Point(int x, int y){
    super(x,y);
  }
}
```

The **Serializable** Interface

- A class **X** that implements the **Serializable** interface tells clients that **X** objects can be stored e.g, on file.
- The interface has no methods

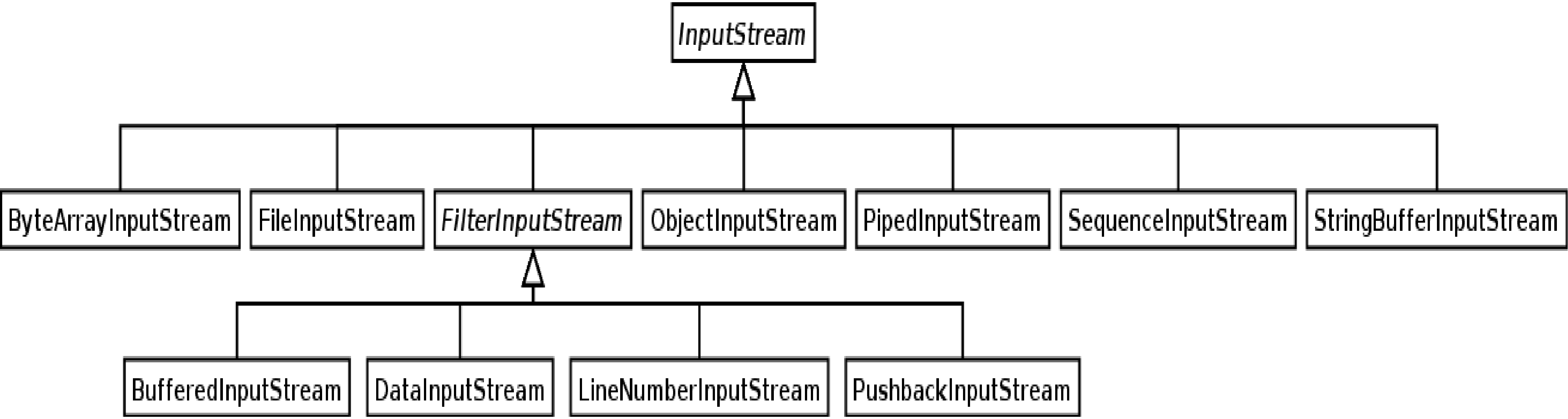
```
public class Car implements Serializable {
  // rest of class unaltered
}
```

The **`Serializable`** Interface, cont.

- Very hard to do in other programming languages!!!
- Class must implement the **`Serializable`** interface
- Uses
  - Output: **`ObjectOutputStream`**
    - **`writeObject()`**
  - Input: **`ObjectInputStream`**
    - **`readObject()`**
- All relevant parts (the web of objects) are serialized.
- Lightweight persistence
  - used in RMI (send objects across a network)
  - used in JavaBeans

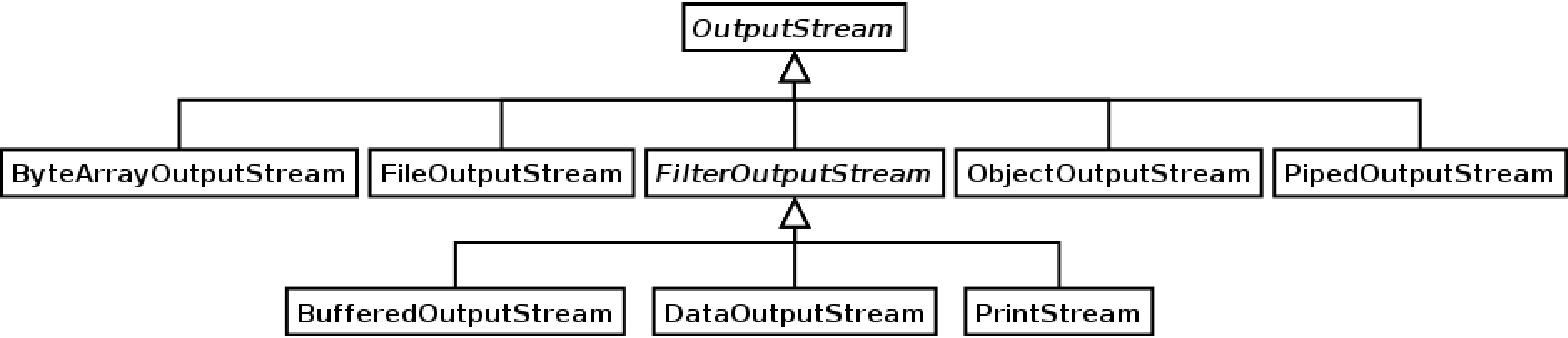
- Similar functionality in C#, PhP, Python, Perl

`InputStream` Hierarchy



- **`InputStream`**, the abstract component root in decorator pattern
- **`FileInputStream`**, etc. the concrete components
- **`FilterInputStream`**, the abstract decorator
- **`LineNumberInputStream`**, **`DataInputStream`**, etc. concrete decorators

OutputStream Hierarchy



- **OutputStream**, the abstract component root in decorator pattern
- **FileOutputStream**, etc. the concrete components
- **FilterOutputStream**, the abstract decorator
- **PrintStream**, **DataOutputStream**, etc. concrete decorators

The **Serializable** Interface, Example

```
// Car class we have seen many times before
import java.io.*;
public class Car implements Serializable { // only change
    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;
    }
   //snip
}
```

The `Serializable` Interface, Example, cont.

```
// Write an object to disk
ObjectOutputStream out =
   new ObjectOutputStream(
      new FileOutputStream("mycars.dat"));

Car myToyota = new Car();
out.writeObject(myToyota);
```

```
// Read an object from disk
ObjectInputStream in =
   new ObjectInputStream(
      new FileInputStream("mycars.dat"));
Car myToyota = (Car)in.readObject();
```

The `Comparable` Interface

- In the package **`java.lang`**.
- Returns

negative integer	if less than
zero	if equals
positive integer	if greater than

```
// 1.4
package java.lang;
public interface Comparable {
    int compareTo(Object o);
}

// 5.0
package java.lang;
public interface Comparable<T> {
    int compareTo(T o);
}
```

The **Comparable** Interface, Example 1.4

```
// 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;
    }
}
```

The **Comparable** Interface, Example 5.0

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

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

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 interface is totally separated from the implementation.
- An interface consists of
  - public abstract methods
  - public constants (public, static, and final variables)
- An interface is a type!
  - return type, formal parameter type, variable type
- Interfaces used throughout the JDK
- Interfaces also in C# and PHP5
  - Not in C++, Perl5, and Python here multiple inheritance