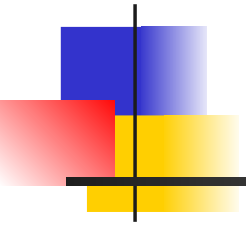


Interfaces

James Brucker

What is an Interface?

An *interface* is a specification of

(1) a required behavior

- any class that claims to "implement" the interface must perform the behavior

(2) constant data values

Example: USB is a specification for an interface.

- precise size of the connector
- electrical: voltage, min/max current, which side provides power
- communications: signaling, protocol, device ID

Any device that means the spec can communicate with other devices -- doesn't matter who manufactured it

Why Use Interfaces?

Separate *specification* of a behavior from it *implementation*

- Many classes can *implement* the same behavior
- Each class can implement the behavior in its own way

Benefits

- The invoker of the behavior is not *coupled* to a *class that implements* the behavior.
- Enables polymorphism and code re-use

Example:

every class can provide its own toString() method. We can print any object without knowing (or being coupled) to its class

Writing an Interface

An interface (before Java 8) may contain only:

- abstract public methods
- public static final variables (constants).

Example:

```
public interface Valuable {
    public static final String CURRENCY = "Baht";
    /** Get the value of this object. */
    public abstract double getValue( );
}
```

abstract method has no method body

You can omit defaults

In an Interface, it is *implied* that:

- all methods are "public abstract"
- all variables are "public static final".

so you don't have to write these. Example:

```
public interface Valuable {
    String CURRENCY = "Baht";
    /** Get the value of this object. */
    double getValue( );
}
```

automatically "public abstract"

Implementing an Interface

A class that implements an interface must implement all the methods in the interface.

It may *use* any static constants in the interface.

```
public class Money implements Valuable {
    private double value;
    /** Get the value of this object. */
    public double getValue( ) {
       return value;
    }
    public String toString() {
       return Double.toString(value)+" "+CURRENCY;
    }
}
```

What an Interface Can/Cannot do

Interfaces can contain:

1. public static final values (constants).
2. public instance method signatures (but no code).
3. "default" implementations of methods (Java 8+).

Interface **Cannot contain**:

- static methods (prior to Java 8)
- non-final or non-static variables
- constructors

Limits on use of Interface

What you can do:

1. declare a class "implements" an interface

2. "implement" more than one interface:

class MyClass implements Runnable, Comparable, Cloneable

3. You can define variables using interface type, and even create an array or ArrayList using an interface as type:

```
// Iterator is an interface with type parameter
Iterator<String> it = wordlist.iterator();
Valuable m = new Money(5, "Baht");
List<Valuable> mlist = new ArrayList<Valuable>();
Runnable[] tasks = new Runnable[5];
```

Interface Cannot ...

You **cannot**:

- create objects of an interface type.
- access static behavior using an interface type.

```
Comparable cmp = new Comparable( );   // ERROR
Valuable m = new Valuable();          // ERROR
```

Practical Example: sorting

Many applications need to **sort** an array of objects.

What we want

- one sort method that can sort (almost) ***anything***.

Question:

what does the sort method needs to know about the objects?

Answer:

needs a way to *compare two* objects to decide which one comes first, ex: is "apple" before "orange" ?

The *Comparable* Interface

```
package java.lang;

interface Comparable {

   int compareTo( Object obj ) ;

}
```

the required behavior

Comparable means two object can be ordered

`compareTo( )` returns the result of comparison:

`a.compareTo( b ) < 0`	a should come before b
`a.compareTo( b ) > 0`	a should come after b
`a.compareTo( b ) = 0`	a and b have same order in a sequence

Arrays.sort( ) for sorting

`Arrays` is a class in java.util.

It has a static **`sort`** method for arrays:

```
String [ ] fruit = { "orange", "grapes", "apple",
                     "durian", "banana" };
Arrays.sort( fruit );

// now print the fruit using "for-each" loop
for( String s : fruit ) System.out.println( s );
apple
banana
durian
grapes
orange
```

Arrays.sort( ) uses *Comparable*

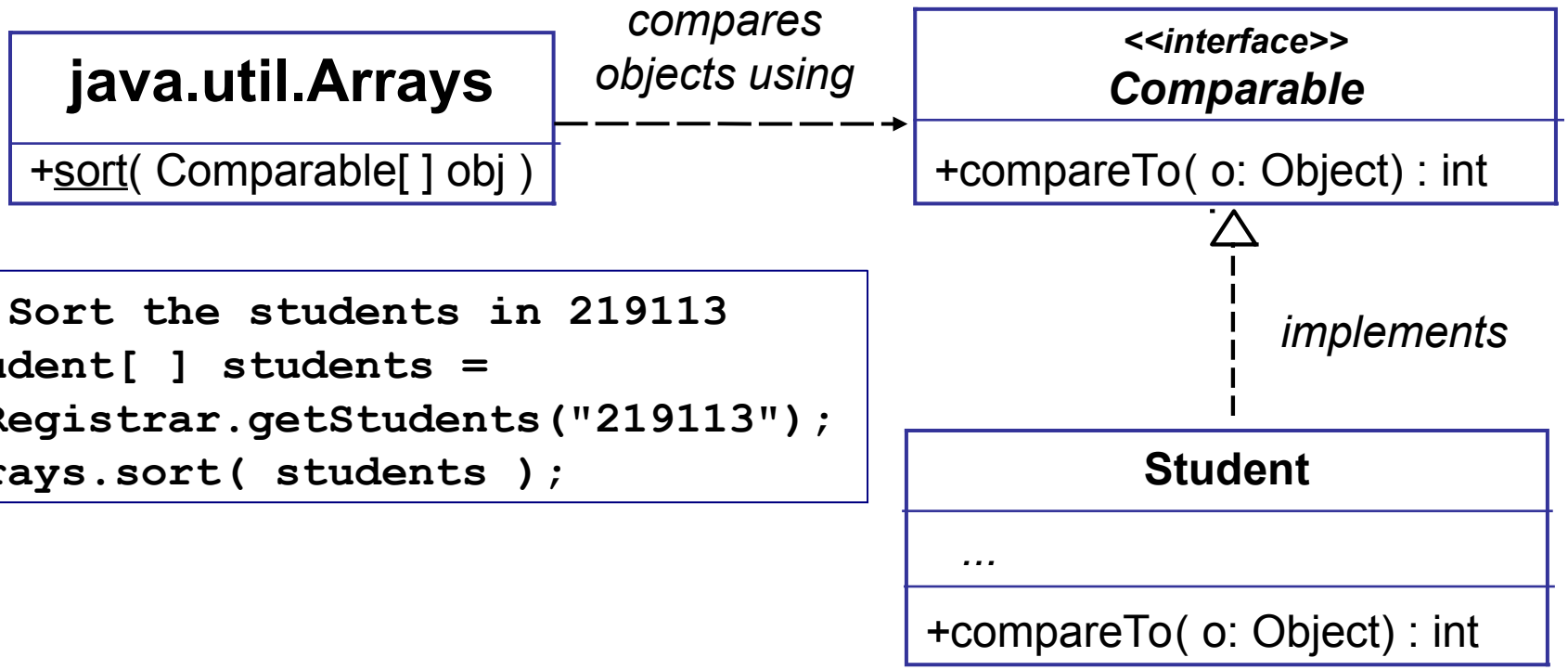
`static void Arrays.sort( Object[ ] a)`

sorts *any kind of object* provides the Objects implement *Comparable*

It ***should be*** : `Arrays.sort( Comparable[] a )`

Comparable enables Code Reuse!

Arrays.sort( ) depends only on the "compareTo" *behavior*, not on any particular class. It can sort anything.




```
// Sort the students in 219113
Student[ ] students =
  Registrar.getStudents("219113");
Arrays.sort( students );
```

Implement the Interface

```
public class Student implements Comparable {
  private String firstName;
  private String lastName;
  private long id;

  public int compareTo( Object obj ) {
      if ( other == null ) return -1;
      /* cast object as a Student */
      Student other = (Student) obj;
      // Compare using Student id
      if ( this.id < other.id ) return -1;
      if ( this.id > other.id ) return +1;
      else return 0;
  }
```

How does Comparable enable Code Reuse?

Sorting:

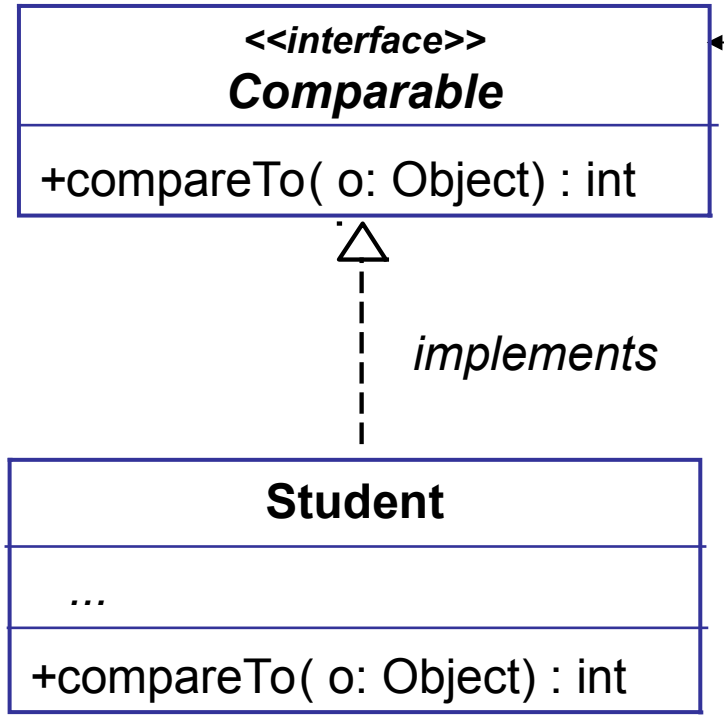
- Any application can use Arrays.sort to sort an array.
- Any application can use Collections.sort to sort a list.

Searching:

- Arrays.binarySearch(array[], key) efficient binary search of an array that has been sorted.
- Collections.binarySearch(collection, key) efficient search of a collection that has been sorted.

UML Notation for Interface

Student implements Comparable



write <<interface>> above the name

You don't need to write "*implements*" on your diagrams,

But you MUST use a dashed arrow and triangle arrowhead as shown here.

Problem with Comparable

A problem of Comparable is that it accepts *any Object* .

- `student.compareTo( dog )`
- `bankAccount.compareTo( string )`

So, you have to do a lot of type checking:

```
public int compareTo( Object other ) {
    if ( !(other instanceOf Student) )
      throw new IllegalArgumentException(". . ." );
    Student st = (Student) other;
    // now compare using Student st
}
```

Parameterized Interfaces

TYPE SAFETY

In Java 5.0+ interfaces and classes can have *type parameters*.

Type parameter is a variable that represent a *type*: the name of a class or interface.

Example: the *Comparable* interface has a type parameter (T) that is a "place holder" for an actual data type.

```
interface Comparable<T> {

   int compareTo( T obj ) ;

}
```

The parameter to compareTo must be this type or a subclass of this type.

Using a Parameterized Interface

Use the type parameter to implement Comparable.

```
public class Student implements Comparable<Student> {
  private String firstName;
  private String lastName;
  private long id;

  public int compareTo( Student other ) {
       /* NO cast or type-check needed */
       //Student other = (Student) obj;
       if ( other == null ) return -1;
       // easy way to compare 2 "long" values
       return Long.compare(this.id, other.id);
  }
```

Here you set the value of the type parameter.

Type Safety

Use a *type parameter* so the compiler can check that you are using **compatible** data types.

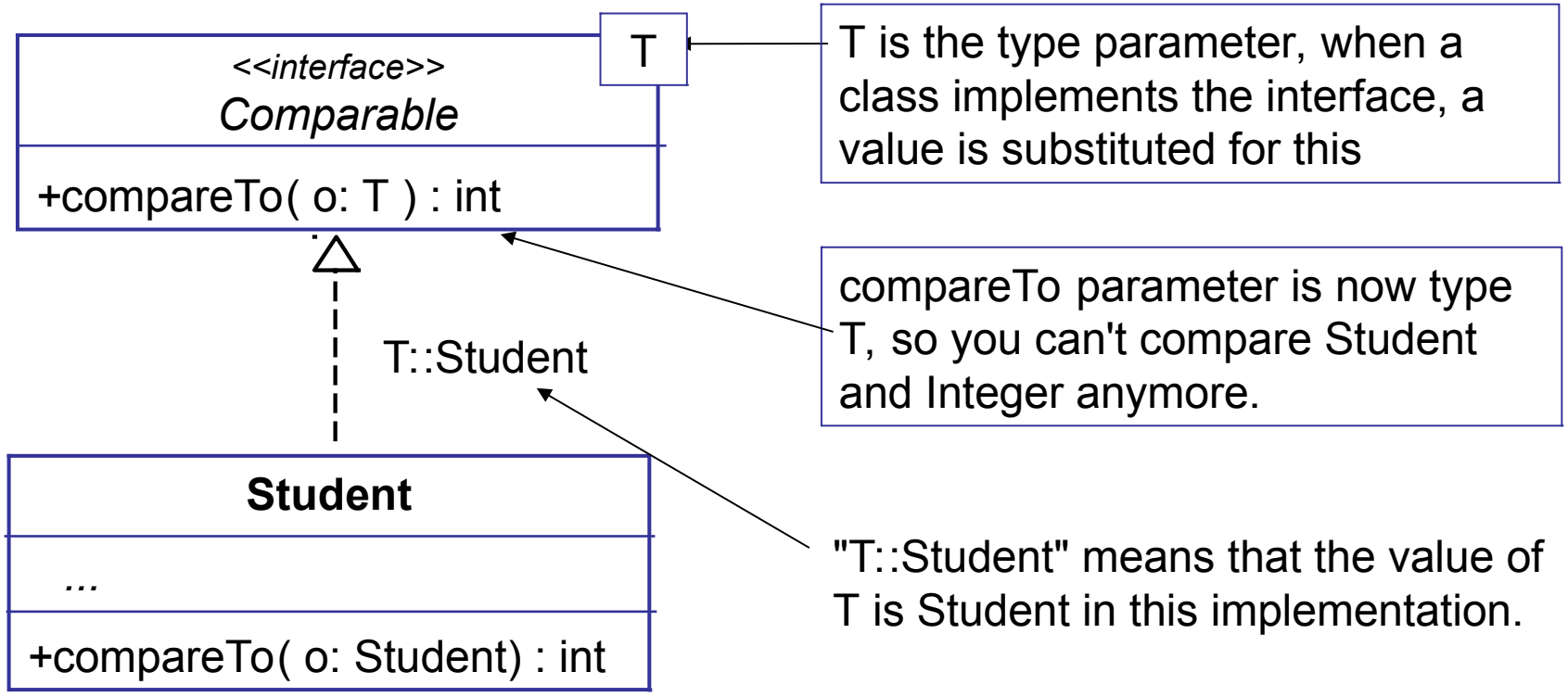
This is called *type safety*.

```
Student joe = new Student("Joe",48541111);
Dog ma = new Dog( "Lassie" );

// compare student to a dog?

if ( joe.compareTo( ma ) > 0 )
  System.out.println("Dog before student");
```

Java compiler will issue an error.

UML for Parameterized Interface

`Student implements Comparable<Student>`



Three Uses of Interfaces

1. Specify a behavior (the most common use).

2. Define constants.

3. *Confirm* a behavior or suitability for purpose.

Use of Interface: *define constants*

- Interface can define public constants.

Example: javax.swing.SwingConstants

```
interface SwingConstants {
  int CENTER = 0;
  int TOP = 1;
  int LEFT = 2;
  int BOTTOM = 3
  int RIGHT = 4;
  ...etc...
```

Fields in an interface are implicitly:

public static final

Accessing *constants from Interface*

```
class MyClass {
  int x = SwingConstants.BOTTOM;
  int y = SwingConstants.LEFT;
```

Many Swing components *implement SwingConstants* so that they have all the SwingConstants.

```
class JLabel extends JComponent
        implements SwingConstants {
  int x = BOTTOM;
  int y = LEFT;
```

Use of Interface: *confirm behavior*

The "Cloneable" interface *confirms* that an object supports the "clone" behavior (deep copy).

```
public class Employee implements Cloneable {
  public Object clone( ) {
 try {
   // first clone our parent object.
   Employee copy = (Employee) super.clone( );
   // now clone our attributes
       copy.firstName = new String( firstName );
       copy.birthday = (Date) birthday.clone( );
       ...etc...
       return copy;
     } catch (CloneNotSupportedException e) {
   return null;
     }
  }
```

clone( ) and Cloneable

If you call clone( ) with an object that does not implement the Clonable interface, Object.clone( ) will throw a CloneNotSupportedException .

```
public class Object {
  ...
  protected Object clone( ) {
    if ( ! (this instanceof Clonable) )
    throw new CloneNotSupportedException();
```

Test for "implements" interface

```
Object x;

if (x instanceof Date)
  // x is a Date or a subclass of Date

if (x instanceof Comparable)
  // x is Comparable
```

Java 8 Interfaces

Java 8 interfaces can contain **code**

1. default methods (instance methods)
2. static methods

Ref: http://java.dzone.com/articles/interface-default-methods-java

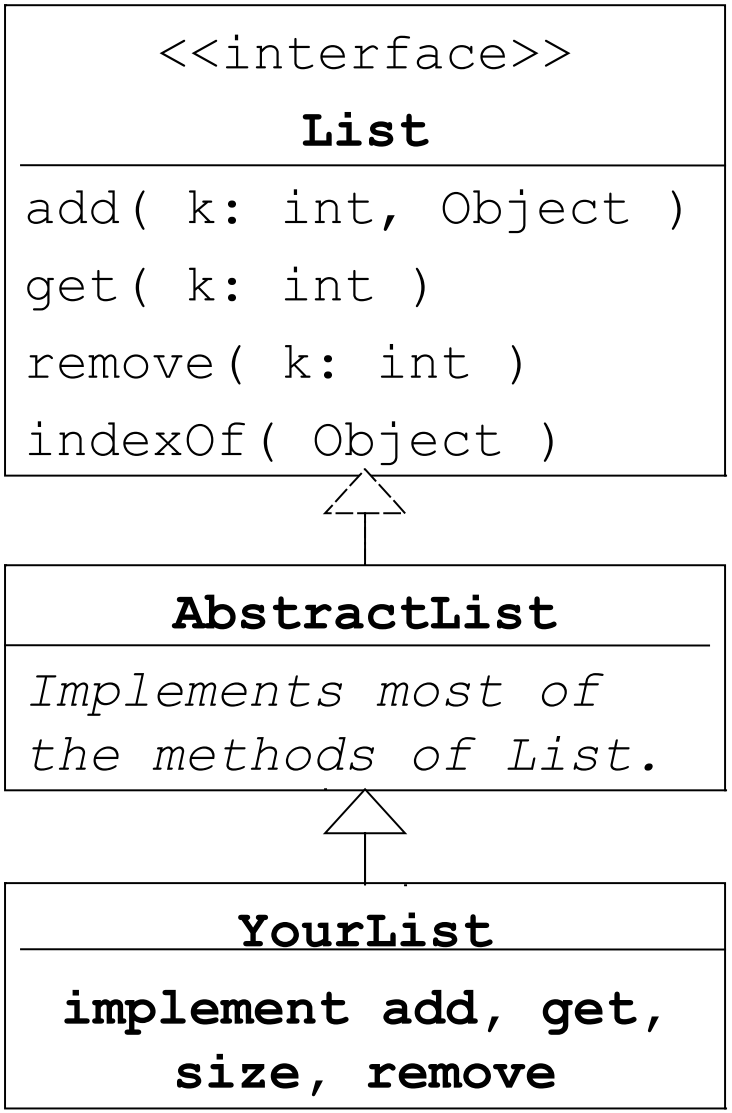
```
public interface Valuable {
    /** Abstract method that must be implemented
     *  by a class.
     */
    public double getValue( );

    /** default method is supplied by interface.
     */
    default boolean isWorthless() {
       return this.getValue() == 0.0;
    }
}
```

Interface and Inheritance

The List interface has 25 methods. Its a lot of work to implement all of them.

AbstractList implements most of the methods for you.

Your List class implements only a few methods.



Summary: interface

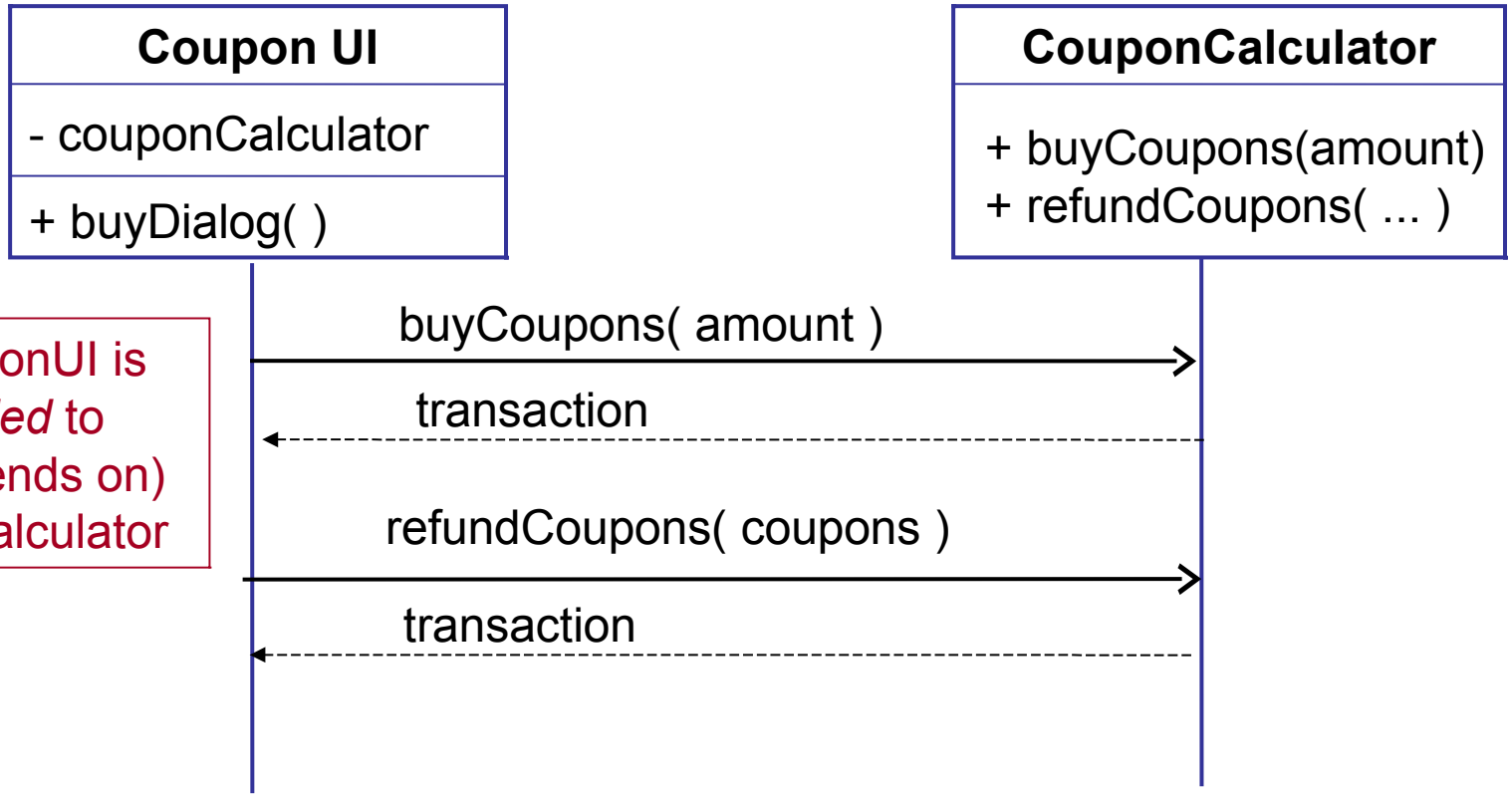
- Interface can contain only:
  - static constants
  - instance method signatures (but no code).
- Implicit properties:
  - all methods are automatically public.
  - all attributes are automatically public static final.

Interface May NOT Contain

- static (class) methods (*allowed in Java 8*)
- implementation of methods (*allowed in Java 8*)
- instance variables
- constructors

Use of Interface: *reduce dependency*

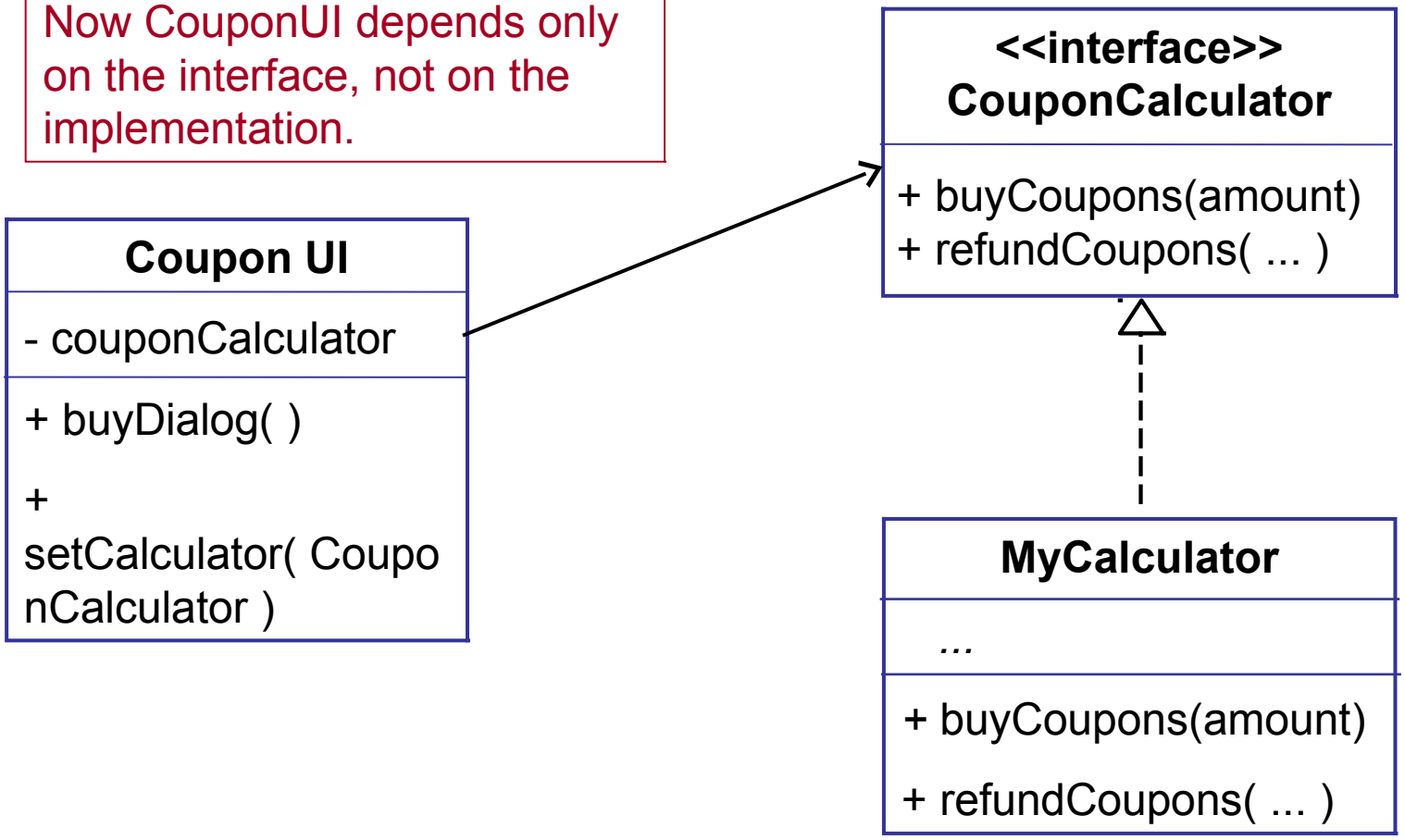
You can eliminate direct dependency between classes by creating an interface for required behavior.



CouponUI is *coupled* to (depends on) the calculator

Use of Interface: *reduce dependency*

Now CouponUI depends only on the interface, not on the implementation.



Example: max(object, object)

Q: Can we write a method to find the max of any two objects?

A: Yes, if objects are *comparable*.

```
/** find the greater of two Comparable objects.
 *  @param obj1, obj2 are the objects to compare.
 *  @return the lexically greater object.
 */
public Comparable max(Comparable obj1, Comparable obj2)
{
 if ( obj2 == null ) return obj1;
 if ( obj1 == null ) return obj2;
 // a.compareTo(b) > 0 means a > b
 if ( obj1.compareTo(obj2) >= 0 ) return obj1;
 else return obj2;
}
```

max(object, object, object)

- How would you find the max of 3 objects?

```
/**
 * find the greater of three objects.
 * @param obj1, obj2, obj3 are objects to compare.
 * @return the lexically greater object.
 */
public static Comparable max( Comparable obj1,
    Comparable obj2, Comparable obj3 )
{




}
```

max(object, object, ...)

- Java 5.0 allows variable length parameter lists.
- One method can accept any number of arguments.

```
/** find the lexically greatest object.
 *  @param arg, ... are the objects to compare.
 *  @return the lexically greatest object.
 */
public Comparable max( Comparable ... arg ) {
 // variable parameter list is passed as an array!
 if ( arg.length == 0 ) return null;
 Comparable maxobj = arg[0];
 for( int k=1; k < arg.length; k++ )
 // a.compareTo(b) < 0 means b "greater" than a.
 if ( maxobj.compareTo(arg[k]) < 0 ) maxobj = arg[k];
 return maxobj;
}
```

How You Can Use Interfaces

- Parameter type can be an interface.
- Return type can be an interface.
- The type of a variable (attrib or local) can be an interface. An array can be a variable.
- As right side (type) of "`x instancof type`".

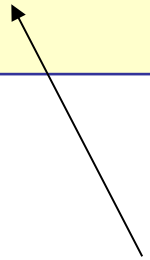
```
public static Comparable max(Comparable [ ] args)
{
   Comparable theBest;
   if ( args == null ) return null;
   theBest = arg[0];
   for( int k = 1; k < args.length ; k++ )
      ... // for you to do
   return theBest;
}
```

OO Analysis & Design Principle

"*Program to an interface, not to an implementation*"

(in this principle, "interface" means a specification)

```
// Purse specifies that the coins
// are a List (an interface type)
public class Purse {

 List<Coin> coins;
```



Comparator

- What if you want to sort some objects, but the class does not implement Comparable?

- or -

- Class implements Comparable, but it isn't the way we want to order the objects.

Example:

We want to sort a list of Strings *ignoring case.*

`String.compareTo( )` *is case sensitive.* It puts "Zebra" before "ant".

Solution: Comparator

The sort method let you specify your own *Comparator* for comparing elements:

Arrays.sort(array[], comparator)

Collections.sort(collection, comparator)

java.util.Comparator:

```
public interface Comparator<T> {
    /**
     * Compare a and b.
     * @return < 0 if a comes before b,
     *         > 0 if a comes after b, 0 if same.
     */
    public int compare(T a, T b);
}
```

Example: Comparator

Sort the coins in reverse order (largest value first):

```
public class CoinComparator
      implements Comparator<Coin> {
   // @precondition a and b are not null
   public int compare(Coin a, Coin b) {
      return b.getValue() - a.getValue();
   }
}
```

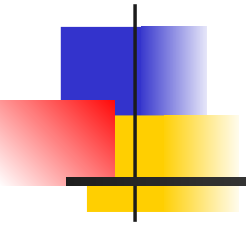
```
List<Coin> coins = ...;
// sort the coins
Comparator<Coin> sorter = new CoinComparator();
Collections.sort( coins, sorter );
```

More Information

- *Core Java, volume 1*, page 223.
- Sun Java Tutorial

Interfaces for "can do"

What it means:	*Interface Name*
Can Run	*Runnable*
Can Compare (two things)	*Comparable*
Can Iterate (hasNext, next)	*Iterable*
Can Clone (make a copy)	*Cloneable*
Can read from	*Readable*



Questions About Interfaces

Multiple Interfaces

Q: Can a class implement multiple interfaces? How?

A:

```
public class MyApplication
                        implements _____?_______ {

   // implement required behavior by Comparable
   public int compareTo( Object other ) { ... }

   // implement behavior for Cloneable
   public Object clone(  ) { ... }
   ...
}
```

Multiple Interfaces

Q: Can a class implement multiple interfaces? How?

A: Yes, separate the interface names by commas.

```
public class MyApplication
             implements Comparable, Cloneable {

  // implement required behavior by Comparable
  public int compareTo( Object other ) { ... }

  // implement behavior for Cloneable
  public Object clone(  ) { ... }
  ...
}
```

Advantage of Interface

Q: What is the advantage of using an interface instead of an Abstract Class to specify behavior?

```
abstract class AbstractComparable {
   /** function specification: no implementation */
   abstract public int compareTo( Object other ) ;
}
```

Abstract method does not have a body.

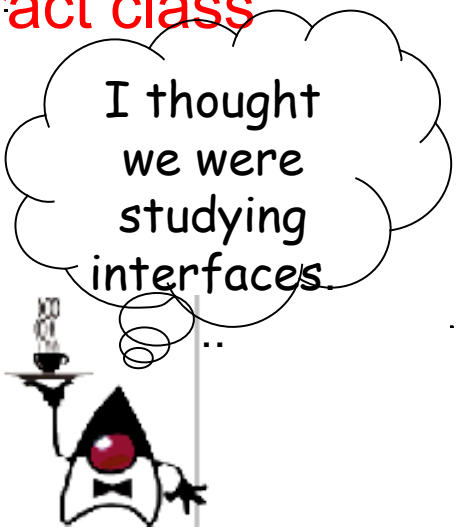
```
public class MyClass extends AbstractComparable {
   /** implement the method */
   public int compareTo( Object other ) {
       ...
    }
}
```

Advantage of Abstract Class

Q: What is the advantage of using an abstract class instead of an interface?

A: An abstract class can provide implementations for some methods.

The client inherits these methods and overrides the ones it wants to change.




```
abstract class AbstractFunction {
   /** function specification: no implementation */
   abstract public double f( double x ) ;
   /** approximate the derivative of f(x) */
   public double derivative( double x ) {
      double dx = 1.0E-12;
      return ( f(x+dx) - f(x-dx) ) / (2*dx);
   }
```

Interface versus Abstract Class (2)

Example: dumb function (no derivative).

```
public class SimpleApp extends AbstractFunction {
  /** actual function */
  public double f( double x ) {return x*Math.exp(-x);}
  // derivativef is inherited from AbstractFunction
}
```

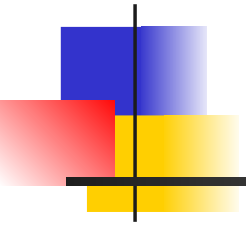
Example: smart function (has derivative).

```
public class BetterApp extends AbstractFunction {
  /** actual function */
  public double f( double x ) {return x*Math.exp(-x);}
  public double derivativef( double x ) {
      return (1 - x) * Math.exp(-x);
  }
}
```

Using Abstract Classes

If we use an abstract class (AbstractFunction) to describe the client, then in the service provider (Optimizer) write:

```
public class Optimizer {
    /** find max of f(x) on the interval [x0, x1] */
   public static double findMax( AbstractFunction fun,
                double x0, double x1 ) {
      double f0, f1, fm, xm;
      f0 = fun.f( x0 );
      f1 = fun.f( x1 );
      do {   xm = 0.5*(x0 + x1);  // midpoint
             fm = fun.f( xm );
             if ( f0 > f1 ) { x1 = xm; f1 = fm; }
             else { x0 = xm; f0 = fm; }
      } while ( Math.abs( x1 - x0 ) > tolerance );
      return ( f1 > f0 ) ? x1 : x0 ;
   }
}
```

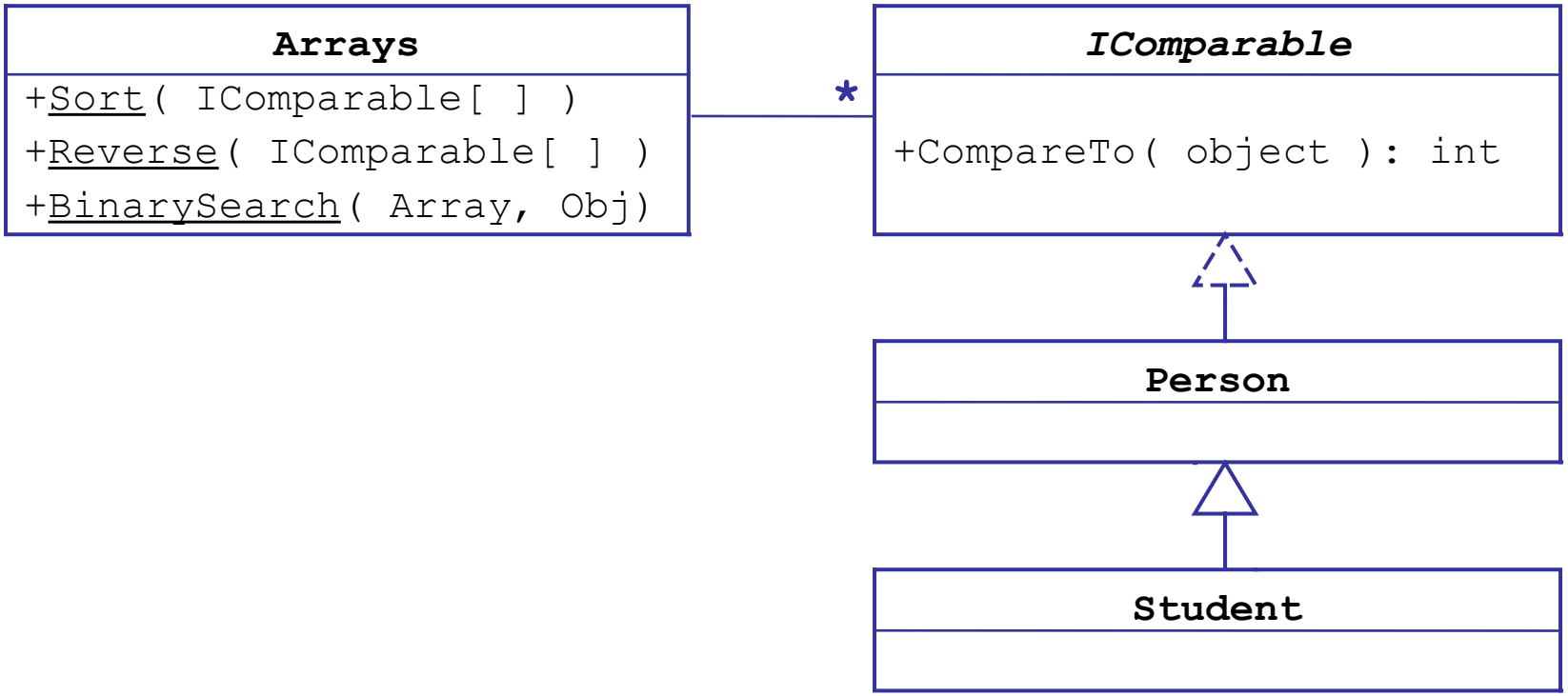


Interfaces in C#

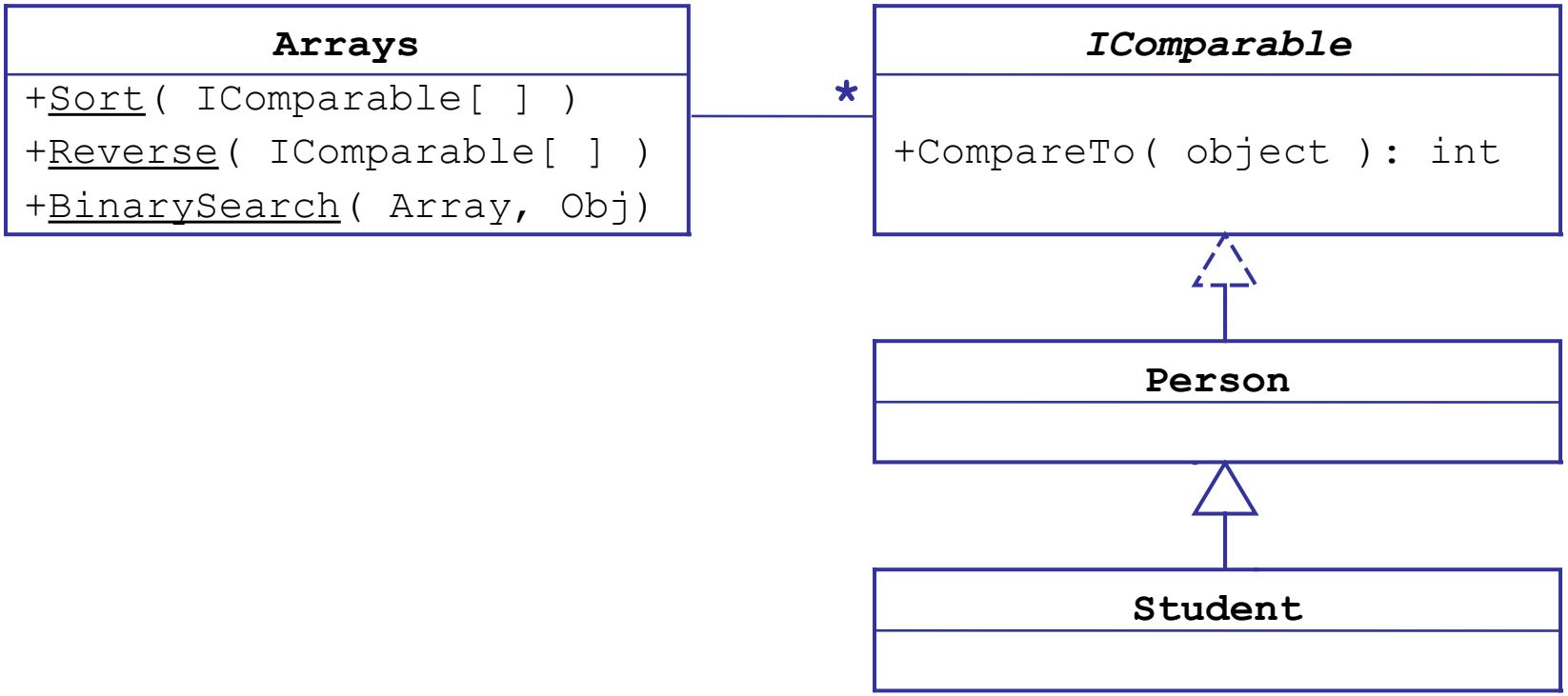
Interface Example in C#

```
/** Person class implements IComparable **/
using namespace System;
public class Person : IComparable {
  private string lastName;
  private string firstName;
  ...
  int CompareTo( object other ) {
    if ( other is Person ) {
      Person p = (Person) other;
      return lastName.CompareTo( p.lastName );
    }
    else throw new IllegalArgumentException(
       "CompareTo argument is not a Person");
  }
}
```

Why implement interface?
What is the benefit?




```
public class Student : Person { .
    private String studentID;
    ...

    // Student is a Person
    // Student inherits CompareTo from Person
}
```




```
Student [] students = course.getStudents( );

// sort the students
Arrays.Sort( students );
```