



Interfaces

James Brucker
Revised 01/20/2015



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 its *implementation*

- ❑ Many classes can *implement* the same behavior
- ❑ Each class can implement the behavior in a way that best suits the class

Benefits

- ❑ The *user* of the behavior is not *coupled* to a particular *provider* of the behavior (the implementation).
- ❑ Enables *polymorphism* and *code re-use*

Example:

any company can design and manufacture a USB device that *implements* the USB interface (specification).



Example: sorting

Many applications want to **sort** an array or list of objects.

The objects to be sorted are part of the application.

What we want

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

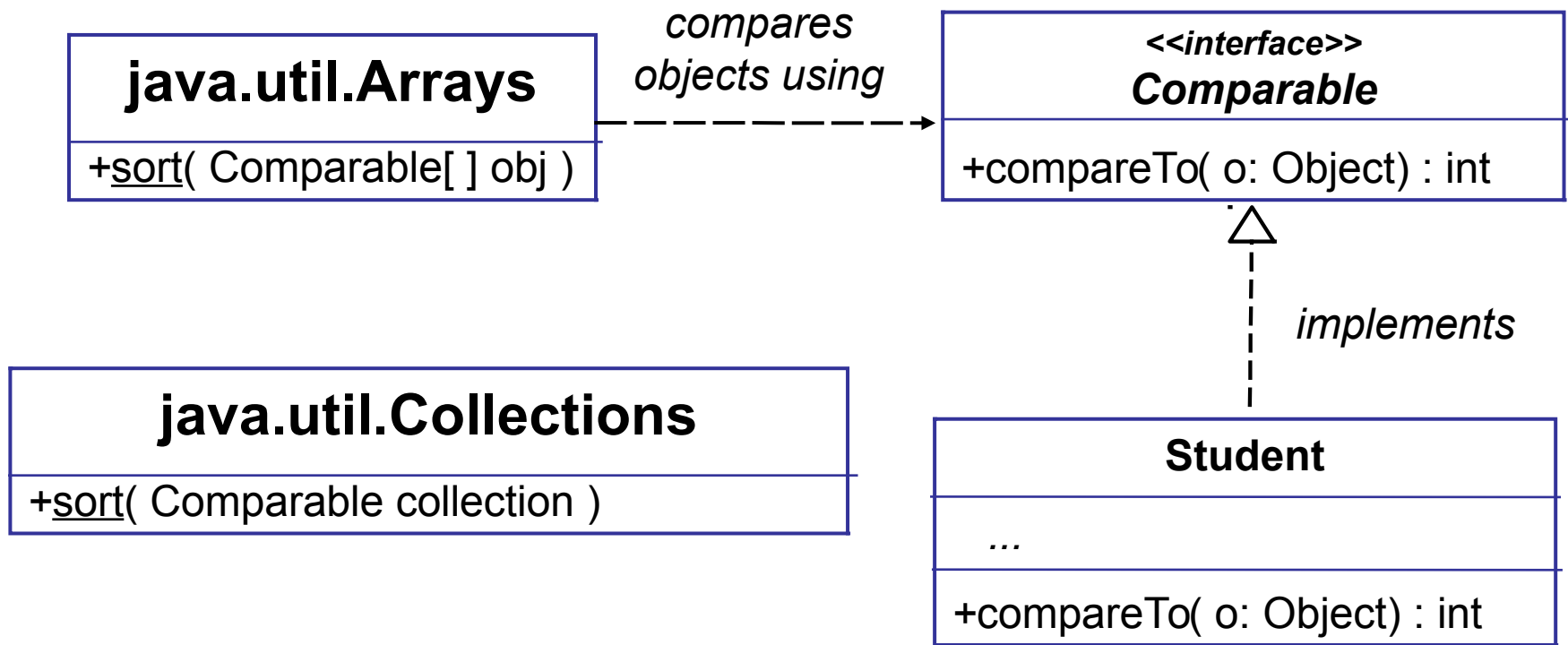
Question:

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

Answer: needs a way to **compare objects** to decide which comes first

How does Comparable enable Code Reuse?

Arrays.sort() *depends only on a behavior*, not on any kind of object (class).





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.



The Java *Comparable* Interface

```
interface Comparable {  
    int compareTo( Object obj ) ;  
}
```

specification of
required behavior

It *specifies* a behavior: objects can be compared and the result is a number (int) with a specified meaning:

$a.compareTo(b) < 0$ 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



Implementing 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 back to an Student */  
        Student other = (Student) obj;  
        // COMPARE IDs  
        if ( ID < other.ID ) return -1;  
        if ( ID == other.ID ) return 0;  
        else return 1;  
    }  
}
```




Example: searching an array

```
class Student implements Comparable<Student> { ... }

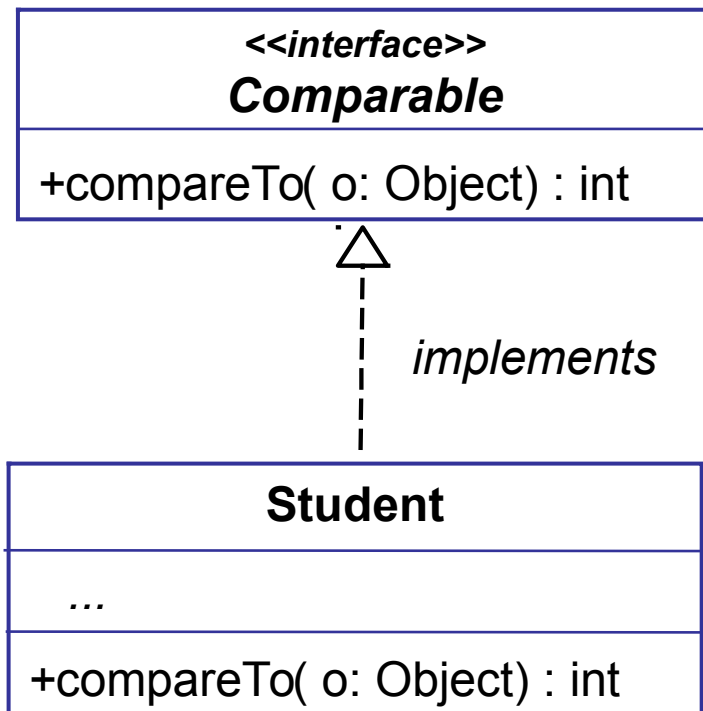
Student [ ] students = registrar.getStudents( "219113" );

// sort the students
java.util.Arrays.sort( students );

// find the student with ID 49541111, in only log(n) time
Student who = new Student( 49541111 );
int k = Arrays.binarySearch( students, who );
if (k >= 0) System.out.println("found student at index "+k);
```

UML Notation for Interface

- Student implements Comparable



← write <<interface>> above the name

You don't need to write "implements" on your UML class diagram, but you MUST use the dashed arrow and triangular arrowhead as shown here.



Comparable and Code Re-use

- `Arrays.sort()` can sort *any kind of objects* -- provided that the objects provide the *Comparable* behavior.
- `Arrays.binarySearch()` can search *any array of objects* -- provided that the objects provide *Comparable*.

Other classes that use *Comparable*:

Calendar, GregorianCalendar, SortedSet, Time



Disadvantage of 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;  
    return this.studentId.compareTo( st.studentId );  
}
```



Parameterized Interfaces

TYPE SAFETY

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

Type parameters are variables 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

You should use the type parameter to implement Comparable.

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

Here you set the value of the type parameter.

```
    public int compareTo( Student other ) {  
        /* NO cast or type-check needed */  
        //Student other = (Student) obj;  
        if ( other == null ) return -1;  
        if ( ID < other.ID ) return -1;  
        if ( ID == other.ID ) return 0;  
        else return +1;  
    }
```



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 dog?
```

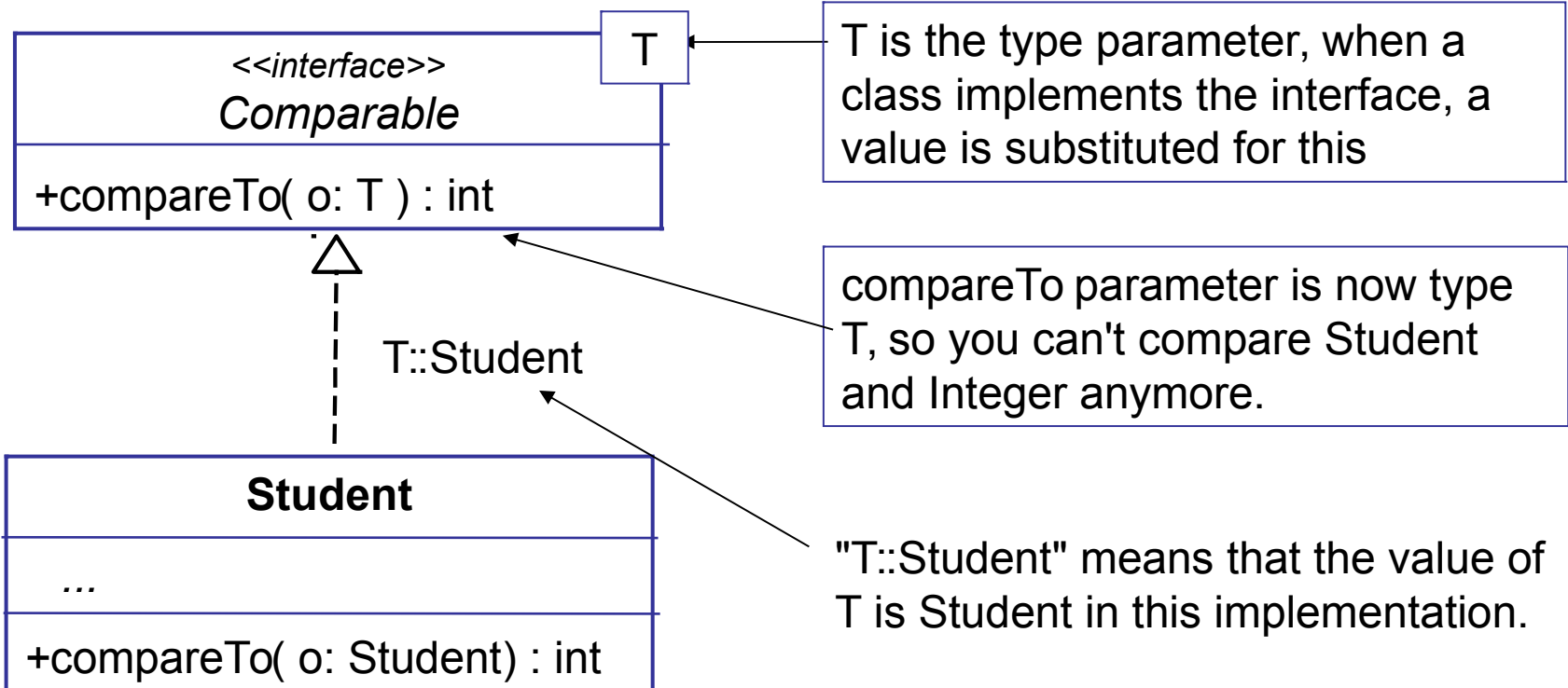
```
if ( joe.compareTo( ma ) > 0 )  
    System.out.println("Dog before student");
```

Java compiler will
issue an error.



UML for Parameterized Interface

Student implements Comparable<Student>





Three Uses of Interfaces

1. Define a behavior (the most common use).
2. Define constants.
3. Confirm a behavior.



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 Cloneable interface, Object.clone() will throw a **CloneNotSupportedException** .

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



Limits on use of Interface

Interfaces can contain:

1. public static constants (final values).
2. public instance method signatures (but no code).

You Cannot:

- ❑ specify static methods (allowed in Java 8)
- ❑ specify non-final variables
- ❑ create objects of interface type.
- ❑ access static behavior using an interface type.
- ❑ specify constructors or instance variables in an interface.

```
Comparable theBest = new Comparable( ); // ERROR
```



Test for 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;  
    }  
}
```




Summary: interface

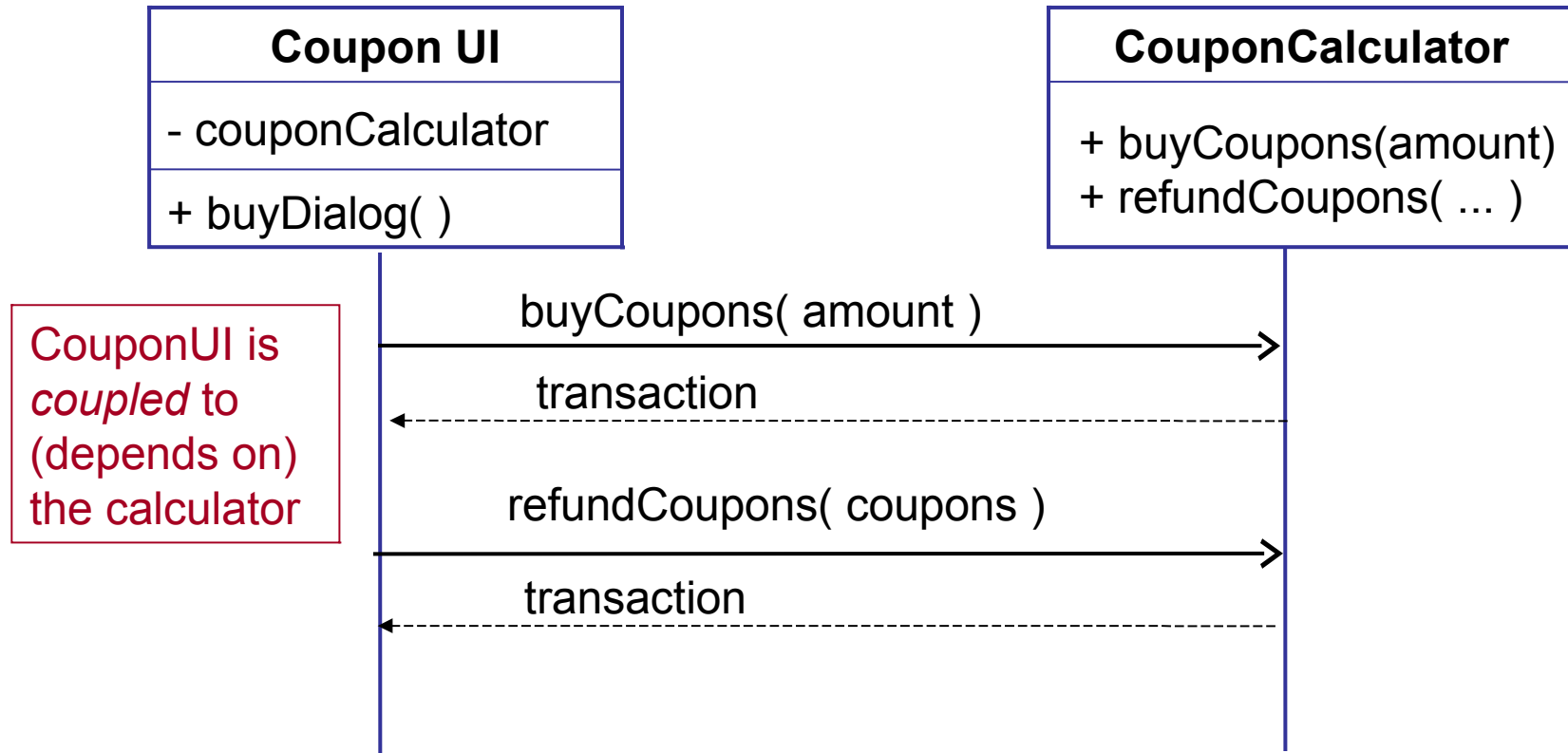
- ❑ Interface can contain only:
 - static constants
 - instance method signatures (but no code).
- ❑ Implicit properties:
 - all methods are automatically **public**.
 - all attributes are automatically **public static final**.

Interface May **NOT** Contain

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

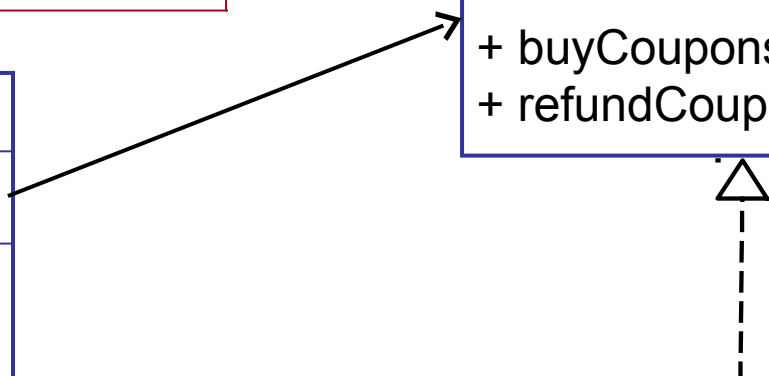
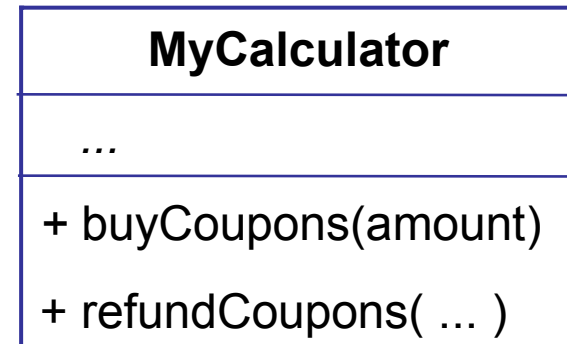
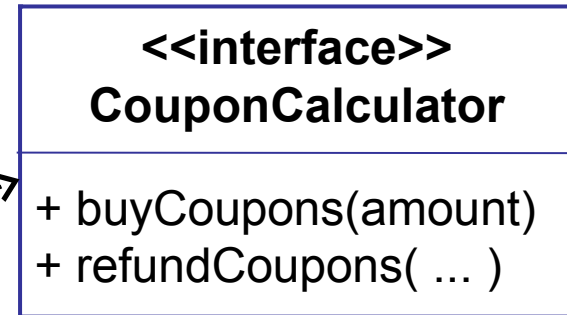
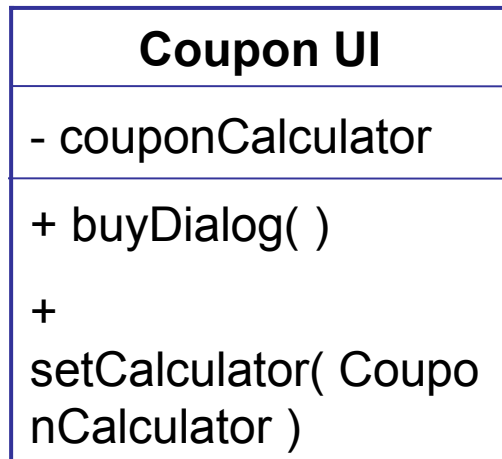
Use of Interface: *reduce dependency*

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



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 instanceof 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;
}
```

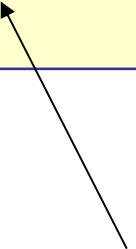


OO Analysis & Design Principle

"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):

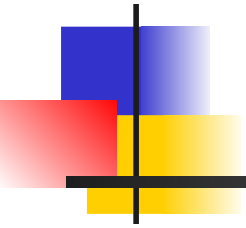
```
public class CoinComparator
    implements Comparator<Coin> {
    // @precondition a and b are not null
    public int compare(Coin a, Coin b) {
        return b.getValue() - a.getValue();
    }
}
```

```
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



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.

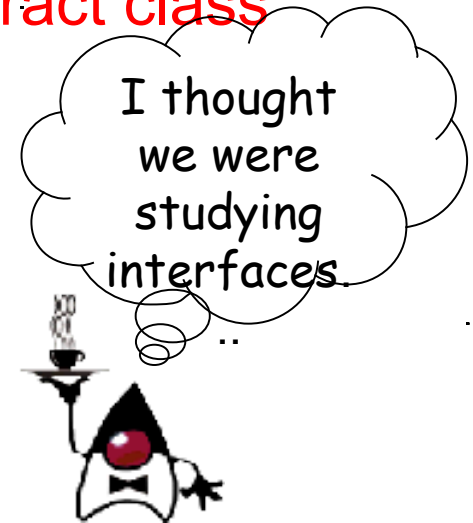
```
public class MyClass extends AbstractComparable {  
    /** implement the method */  
    public int compareTo( Object other ) {  
        ...  
    }  
}
```


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 ;
    }
}
```



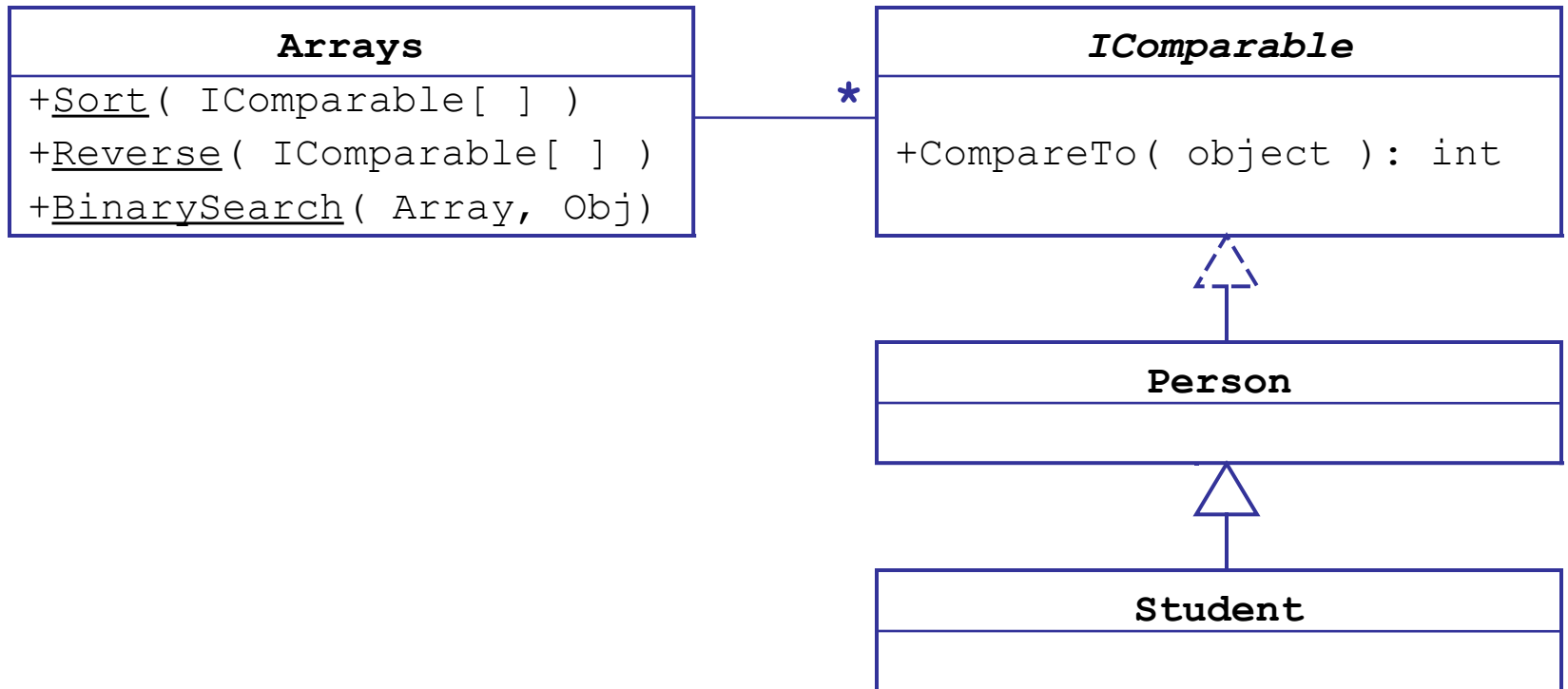
Interfaces in C#



Interface Example in C#

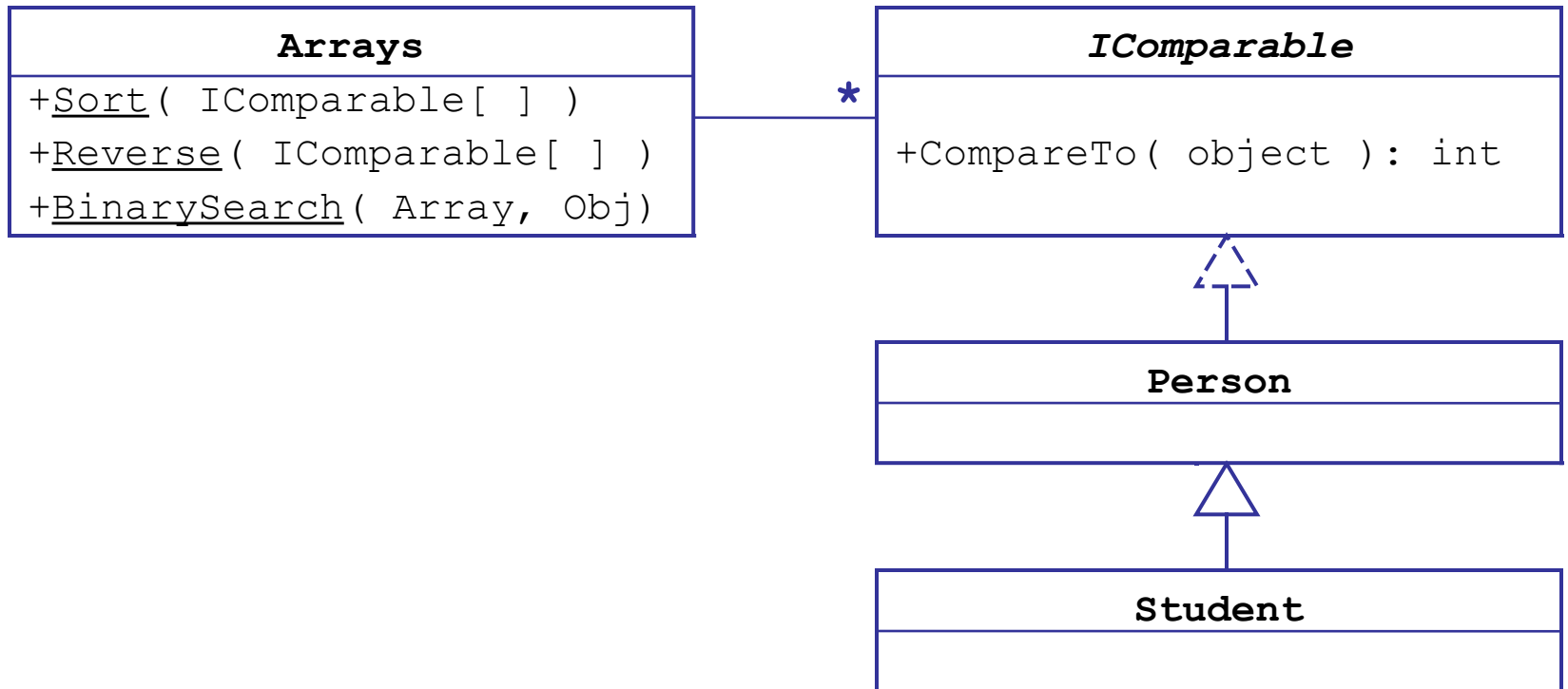
```
/** Person class implements IComparable */  
using namespace System;  
public class Person : IComparable {  
    private string lastName;  
    private string firstName;  
    ...  
    int CompareTo( object other ) {  
        if ( other is Person ) {  
            Person p = (Person) other;  
            return lastName.CompareTo( p.lastName );  
        }  
        else throw new ArgumentException(  
            "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 );
```