Interfaces

ACS-1904 LECTURE 6

Interfaces

- An interface is not a class
  - It’s like a class
  - It’s a set of requirements that classes who want to implement the interface must meet

Interfaces

- Interfaces are not instantiated directly
  - There is never a statement like
    - x = new SomeInterface(…); // not allowed
    - It is, however, acceptable to declare an interface type variable
    - SomeInterface x; // allowed
  - Now x can refer to some object that implements the SomeInterface interface

Interfaces vs. Abstract classes

- Interfaces and Abstract classes can't be instantiated directly
- An interface is 100% abstract
  - No instance fields
  - No instance methods* (the asterisk indicate that there's a bit more to this story)

Full disclosure:

- Interfaces can define constants and they may include static and private methods
- But, of course, these methods can't refer to instance fields, there are none.

Interfaces vs. Abstract classes

- An abstract class is only partially abstract (ironically)
  - It can have both abstract and concrete methods and it can have instance variables.
- A class can extend at most 1 and only 1 abstract class
- A class can implement many interfaces.
- Sidenote: All method signatures in an Interface are automatically public so we don’t need tag them as public.
- All fields in an Interface are public static final

Interfaces

- Interfaces are used heavily in the Java class libraries
  - We will examine `Comparable` and `Comparator`
- An interface is implemented by one or more *concrete* classes
  - Objects of that class can be instantiated
  - Those objects are of the class, and also of the interface type
  - Remember the ”is a” relationship?

Interfaces

Topics

- The `Comparable` interface – the mechanism for providing a natural ordering of objects
  - E.g. `String` implements `Comparable`
- Defining our own interfaces
- The `Comparator` interface – a mechanism to provide alternate orderings for objects

Interfaces - Comparable and String

- The `Comparable` interface includes one method named **`compareTo`**
- `compareTo` compares two strings, say `a` and `b`: **`a`**`.compareTo(`**`b`**`)`

  and returns:

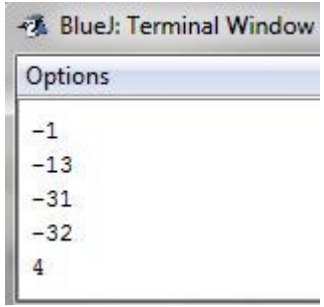
  - negative `int` if `a < b`
  - positive `int` if `a > b`
  - zero if `a` equals `b`

- We say `Comparable` defines the *natural ordering* for objects of an implementing subclass
- `String` is one example of a class that implements `Comparable`
- Strings are ordered lexicographically (roughly speaking - dictionary order)

Example(StringCompareto.java)

```
public class StringCompareto {
    public static void main (String[] args){
        System.out.println("123".compareTo("124"));
        System.out.println("124".compareTo("12A"));
        System.out.println("124".compareTo("PROGRAM"));
        System.out.println("PROGRAM".compareTo("program"));
        System.out.println("programming".compareTo("program"))
    ;
```

Output from the above

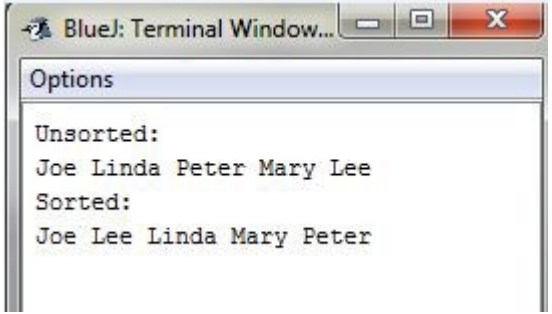


`compareTo` gives us the natural ordering for strings

Note: `Arrays.sort()` and `Collections.sort()` utilize the `compareTo` method for the objects they sort

Example(SortingStrings.java)

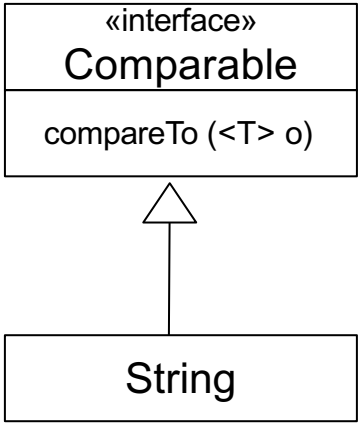
```
public class SortingStrings
{
   public static void main (String[] args){
       String [] names =
       {"Joe", "Linda", "Peter", "Mary", "Lee"};

    System.out.println("Unsorted:");
       for (String n: names)
          System.out.print(n+" ");

       Arrays.sort(names);

       System.out.println("\nSorted:");
       for (String n: names)
           System.out.print(n+" ");
```



Output

`sort` reorganizes the array of strings

In a UML diagram we can illustrate the `Comparable` interface, its method, and that `String` implements `Comparable`



The `Comparable` interface has one method, `compareTo`

String is a class implementing `Comparable`

So String has a `compareTo` method

Comparable - Java documentation

java.lang

Interface Comparable<T>

Type Parameters:

T - the type of objects that this object may be compared to

All Known Subinterfaces:

ChronoLocalDate, ChronoLocalDateTime<D>, Chronology, ChronoZonedD

All Known Implementing Classes:

AbstractChronology, AbstractRegionPainter.PaintContext.CacheMode, Authenticator.RequestorType, BigDecimal, BigInteger, Boolean, Byt

For example, `T` is `String`

`String` is included here

`String` must have an implementation of `compareTo`

```
public interface Comparable<T>
```

This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's *natural ordering*, and the class's `compareTo` method is referred to as its *natural comparison method*.

Lists (and arrays) of objects that implement this interface can be sorted automatically by `Collections.sort` (and `Arrays.sort`). Objects that implement this interface can be used as keys in a sorted map or as elements in a sorted set, without the need to specify a comparator.

compareTo

```
int compareTo(T o)
```

Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

`compareTo` returns +ve, -ve or 0

String - Java documentation

compareTo

```
public int compareTo(String anotherString)
```

CompareTo entry in String

Compares two strings lexicographically.

Description of compareTo

Returns:

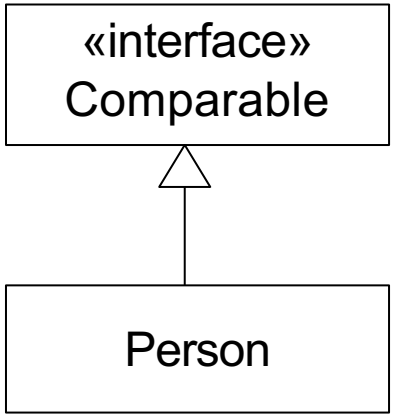
the value 0 if the argument string is equal to this string; a value less than 0 if this string is lexicographically less than the string argument; and a value greater than 0 if this string is lexicographically greater than the string argument.

What compareTo returns

Making a class comparable

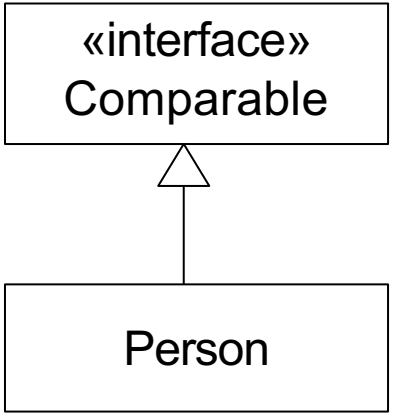
- Situation: suppose we have a class, `Person`, and we need to be able to sort objects of type `Person`
  - We can specify `Person` implements `Comparable`, code a `compareTo` method

    → we can sort them using either `Arrays.sort` or `Collections.sort`



Person.java

```java
public class Person implements Comparable<Person> {
    private String name;
    private int year;  // year of birth
    public Person(String n, int y) {
        name = n;
        year = y;
    }
    public int compareTo (Person p) {
        // compare the name of this object to the name of object p
        return name.compareTo(p.name);
    }
    public String getName(){
        return name;
    }
    public void setName(String n){
        name = n;
    }
    public int getYear(){
        return year;
    }
    public void setYear(int y){
        year = y;
    }
    public String toString(){
        return name;
    }
}
```

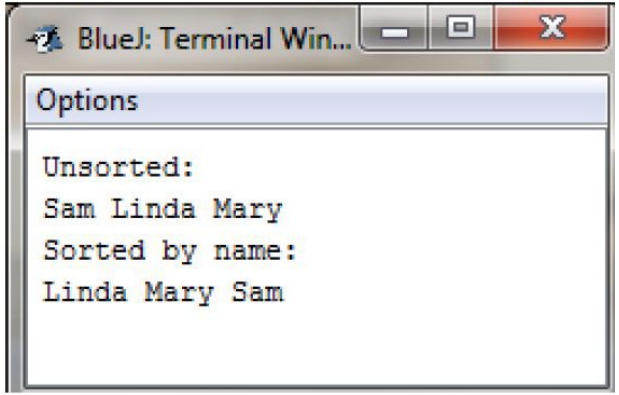


SortingPersons.java

```
import java.util.Arrays;
public class SortingPersons
{
   public static void main (String[] args){
       Person [] people = {
           new Person("Sam",1972),
           new Person("Linda", 1974),
           new Person ( "Mary", 1957)};
       System.out.println("Unsorted:");
       for (Person p: people)
          System.out.print(p+ " ");

       Arrays.sort(people);

       System.out.println("\nSorted by name:");
       for (Person p: people)
          System.out.print(p+" ");
    }
}
```

`Person` implements `Comparable` and so an array of `Person` objects can be sorted

Figure 6.4: Sorting Persons output.



Developing our own interface

Situation:

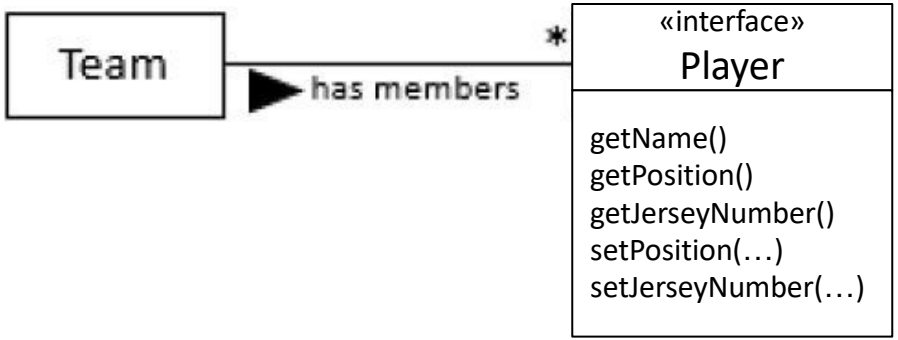
Developing software for teams, players, …

- A team comprises several players
- If we let `Player` be an interface then we have choices for its implementation
- So, we'll start with `Player` as an interface

Developing our own interface - Player interface

`Player` interface specifies methods

- things that a player can know and do

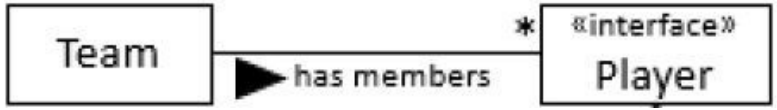


Now, we can write code for Team →

Developing our own interface - Team class

Team.java

```
import java.util.ArrayList;
/** A team comprises several players */
public class Team
{
    private String teamName;
    private ArrayList<Player> members;
    public Team(String name){
        teamName = name;
        members = new ArrayList();
    }

    public void addPlayer (Player p){
        members.add(p);
    }

    public void display (){
        System.out.println(teamName);
        for (Player p: members)
            System.out.println(p.getName()
                +" \t"+ p.getJerseyNumber()
                +" \t"+ p.getPosition());
    }
}
```



Team comprises players

We can add players to the team

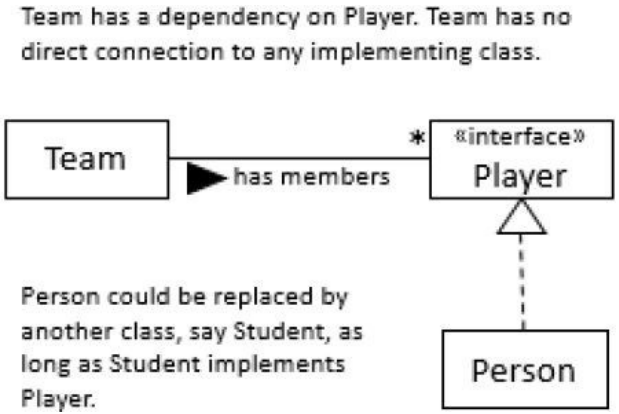
We can display the team

Developing our own interface - Player class

At some time we can define a class that implements `Player`

We can choose that this will be `Person`

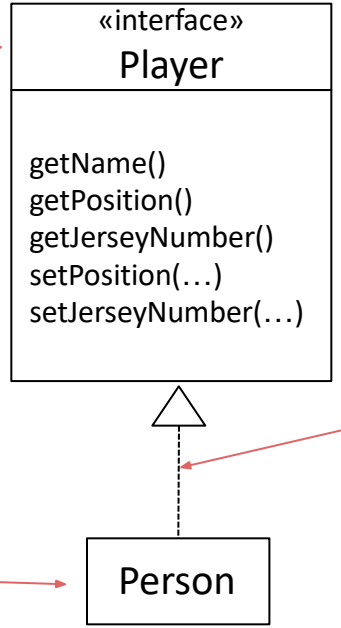
- Add some fields, methods



Developing our own interface - Player interface Player.java

Player is an interface

Player includes methods



Person is a class

The dashed line means that Person *implements* Player

The code for `Person` must include `getName()… and all of the other methods in Player`

Developing our own interface - Person class

Person.java

```
public class Person
    implements Comparable<Person>, Player {
    private String name;
    private int year; // year of birth
    private String position;
    private int jerseyNumber;

    public Person(String n, int y) {
        name = n;
        year = y;
    }
    // methods of the Player interface
    public String getName(){
        return name;
    }
    public String getPosition(){
        return position;
    }
    public int  getJerseyNumber(){
        return jerseyNumber;
    }
    public void setPosition(String p){
        position = p;
    }
    public void setJerseyNumber(int j){
        jerseyNumber = j;
    } ...
```

New fields

Code required for Player interface

Rest of Person

Developing our own interface - Creating/displaying a team

CreateTeam.java

```
public class CreateTeam {
    private Team team;
    public static void main (String[] args) {
        // create 5 persons but know them as Players
        Player p0 = new Person("Jim",1978);
        p0.setPosition( "Centre" );
        p0.setJerseyNumber(1);
        Player p1 = new Person("Sue",1962);
        p1.setPosition( "Left Wing");
        p1.setJerseyNumber(3);
        Player p2 = new Person("Sam",1975);
        p2.setPosition( "Right Wing" );
        p2.setJerseyNumber(8);
        Player p3 = new Person("Tom",1975);
        p3.setPosition( "Left Defence" );
        p3.setJerseyNumber(4);
        Player p4 = new Person("Deb",1966);
        p4.setPosition( "Right Defence" );
        p4.setJerseyNumber(9);
        // create a team
        Team rr = new Team("Red River CC");
        // add players to the team and display
        rr.addPlayer(p0);
        rr.addPlayer(p1);
        rr.addPlayer(p2);
        rr.addPlayer(p3);
        rr.addPlayer(p4);

        rr.display();
```

Each player is created as a person, but since `Person` implements `Player`, a person can be referenced as a `Player`

Players added to the team

Team numbers are displayed

Comparator interface - provides an alternate ordering

The `Comparator` interface has one method `compare()`

- `compare()` has two parameters
  - there are 2 objects to be compared.
- `compare()` compares two objects and returns an `int` value that is

negative	object 1 < object 2
positive	object 1 > object 2
zero	object 1 equals object 2

The OrderByYear comparator (OrderByYear.java)

To provide an ordering for Person objects based on year-of-birth

- To compare two Person objects by year of birth

Class Name is OrderByYear

This is a comparator

This comparator works with Person objects

```
import java.util.Comparator;

public class OrderByYear implements Comparator<Person>
{
    public int compare (Person o1, Person o2) {
        return o1.getYear()-o2.getYear();
    }
}
```

The compare method requires two arguments … two objects to be compared

This comparator is using the year of birth and returning a value: Negative, Positive, Zero

E.g. Comparator with sort( )

SortingPersonsByYearWithComparator.java

`sort()` can be passed two arguments : the array/list and a comparator object

```
import java.util.Arrays;
public class SortingPersonsByYearWithComparator
{
   public static void main(String[] args){
       Person [] people = {
           new Person("Mary", 1957),
           new Person("Terry",1972),
           new Person ( "Zeke", 1957),
           new Person("Sammy",1972),
           new Person("Linda", 1971)};
       // sort with a comparator
       Arrays.sort(people, new OrderByYear());
       for (Person p: people)
          System.out.println(p.getYear()+ " "+p);
    }
}
```

An array of persons

sort() called with a comparator object

Summary

We have examined the `Comparable` interface

- `String` has a `compareTo` method
- `Arrays` and `Collections` have a `sort(…)` method
- The `sort` method uses the `compareTo` method when it sorts objects
- `String` implements `Comparable`

We have developed an interface

- `Team` consists of players → `Player` interface
- `Person` implemented `Player`
- A class can implement any number of interfaces

We have examined the `Comparator` interface

- `Comparator` has one method, `compare`
- A way to provide alternate orderings for objects
  - `OrderByYear` is a comparator for `Person` objects , based on year-of-birth