COMPSCI 121: JAVA INTERFACES

SPRING 2020

GOALS FOR TODAY'S CLASS

We now add another piece to the Object-Oriented Programming (OOP) paradigm.

We know:

- A *class* provides data/behaviors for objects.
- *Inheritance*: a subclass specializes superclass behaviors.
- An *Abstract* class specifies required behavior of its subclasses.
- Interfaces:

EXAMPLE OF AN INTERFACE- SOFTWARE FEATURES

A features of any text editor app: *copy*, *cut* & *paste*.

- User only knows how to use its functionality i.e. its interface.
- Implementation is completely opaque to user.
- Don't need or want to know how it is *implemented*.
- An implementer *may alter* the implementation at any time- User doesn't know (e.g. software updates, bug fixes, etc.).

RECALL: JAVA INHERITANCE

Inheritance by **Extension:**
Use existing code (attributes and functionality in a superclass) but specialize, or extend, it in some way.
Abstract classes ensure subclasses must implement any methods defined as **abstract** in their superclass.

TODAY:
Inheritance by **Implementation**:
An **Interface** specifies a set of methods that an implementing class must provide a definition for.

INTERFACE- COMPARE TO ABSTRACT CLASS

An interface is similar to an abstract class:

- They both provide a set of methods which must be implemented by subclasses.
- They cannot be instantiated.

An interface is different from an abstract class:

- An interface does not provide any implementation.
- An abstract class does provide some implementation.
- Subclasses can extend only one abstract class but can implement many interfaces.

THE CONCEPT OF AN INTERFACE- JAVA CLASS

All Java classes have an interface- their set of public methods.
That determines how the class can be used.
But, the implementations are not visible to the user.

User:
Calls the
`tan`
method.

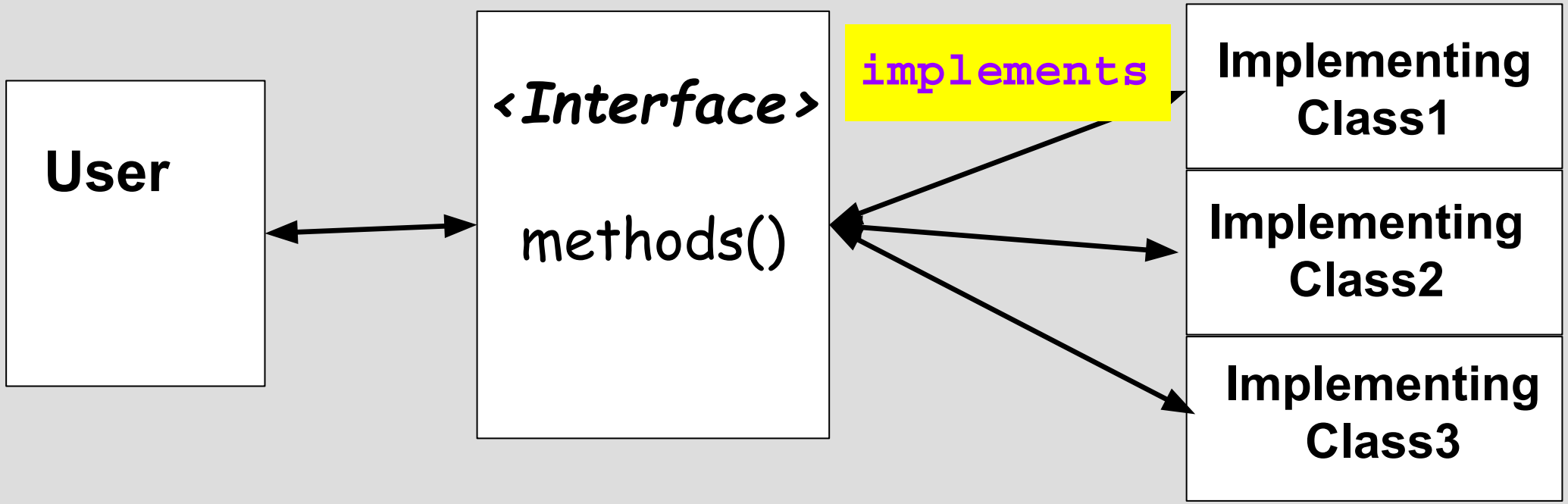
```
public class Calculator {

public static double tan(double angle)
```

- The User does not have access to the implementation of the `tan` method.
- The User is not concerned about how it works, just that it provides a correct result.
- Note that there is only *one implementation* of the `tan` method.

THE CONCEPT OF AN INTERFACE- JAVA INTERFACE DEFINITION

The Java Interface allows for *many ways* to implement a set of methods.
The implementation is NOT exposed to the User.

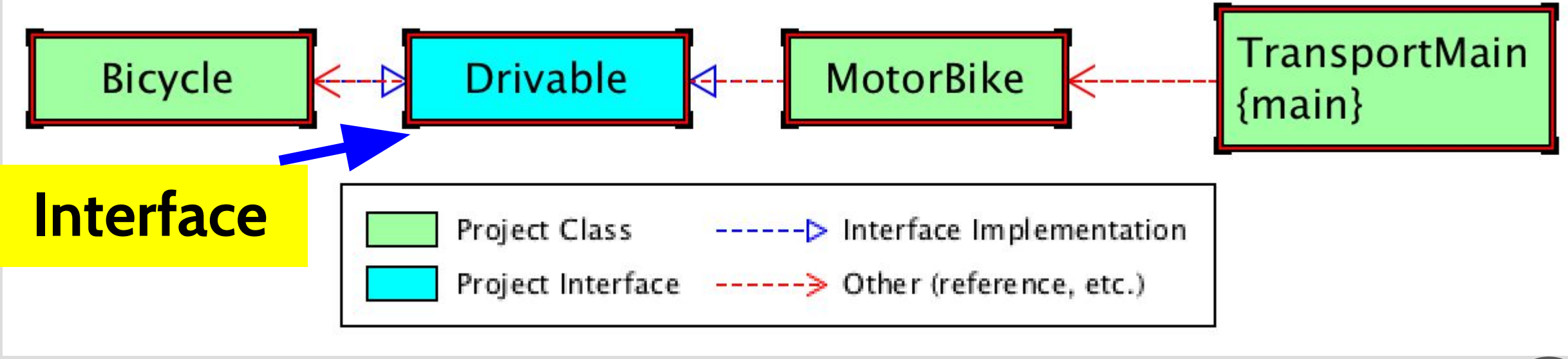


- Allows for changes in implementation to be made without affecting any other parts of the code base.
- The relationship between Interface and implementing classes is `implements` (different than `extends`).

THE SYNTAX OF A JAVA INTERFACE

NOTE:

- Cannot instantiate interface - does not have a constructor.
- All methods are public and abstract.
- May contain only static and final member fields.



JAVA SYNTAX FOR IMPLEMENTING CLASS

```
public class MotorBike implements Drivable {
    int speed;
    int gear;

    public MotorBike(){
      speed = 1;
      gear = 1;
    }

    @Override // to change gear
    public void changeGear(int newGear){

        gear = newGear;
    }

    @Override // to increase speed
    public void speedUp(){

        speed = speed * 2;
    }

    @Override // to decrease speed
    public void applyBrakes(int decrement){

        speed = speed - decrement;
    }
```

implements

Implementing class must implement interface's methods.

```
1 public interface Drivable {
2     public void changeGear(int a);
3     public void speedUp();
4     public void applyBrakes(int a);
5     public void printStatus();
6 }
```

MotorBike must override the Drivable Interface's methods.

Clicker Question 1

```
(1) public interface SomethingIsWrong {
      public void aMethod(int aValue);}

(2) public interface SomethingIsWrong {
      public abstract aMethod(int aValue){
        System.out.println("Hi");}}

(3) public interface SomethingIsWrong {
       public aMethod(int aValue);}
```

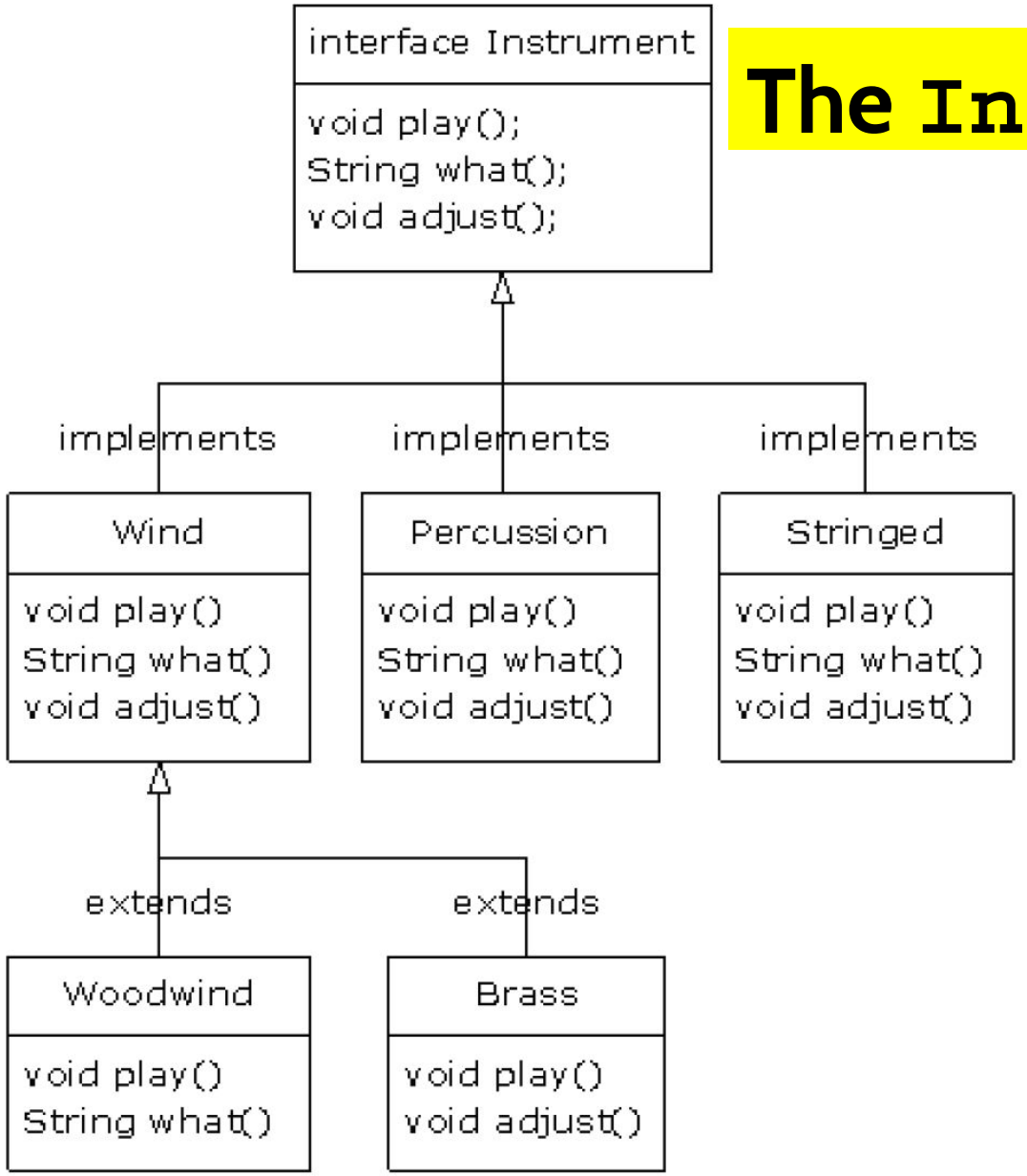
A. All are correct
B. All are incorrect
C. Only (1) is correct
D. Only (2) is correct
E. Only (3) is correct

Clicker Question # Answer 1

```
(1) public interface SomethingIsWrong {
      public void aMethod(int aValue);
    }
(2) public interface SomethingIsWrong {
      public abstract aMethod(int aValue){
        System.out.println("Hi");}
     }
(3) public interface SomethingIsWrong {
      private int aMethod(int aValue);
     }
```

A. All are correct
B. All are incorrect
C. Only (1) is correct Method does not have implementation
D. Only (2) is correct Cannot have implementation
E. Only (3) is correct Cannot be private method

Clicker Question 2

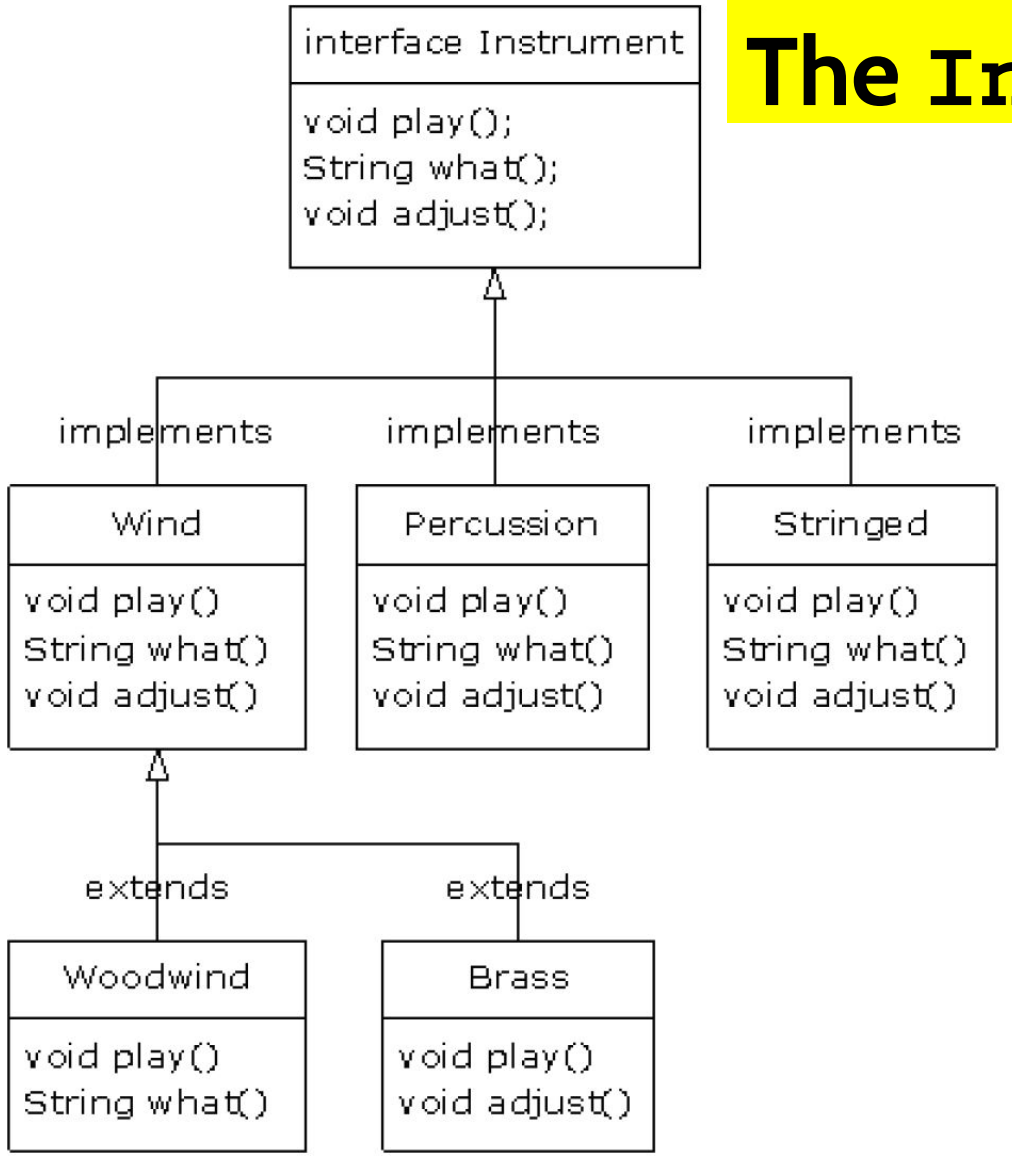


The `Instrument` interface

1. `Instrument` provides an API that must be implemented.
2. `Instrument` provides no other code.
3. All methods in `Instrument` are public.
4. An interface has no method bodies

A. All are correct
B. All are incorrect
C. Only (1) is correct
D. Only (2) is correct
E. Only (3) is correct

Clicker Question 2 Answer



The `Instrument` interface

1. `Instrument` provides an API that must be implemented.
2. `Instrument` provides no other code.
3. All methods in `Instrument` are public.
4. An interface has no method bodies

A. All are correct
B. All are incorrect
C. Only (1) is correct
D. Only (2) is correct
E. Only (3) is correct

INTERFACE USAGE EXAMPLE: JAVA'S *COMPARABLE* INTERFACE

Problem: We have an array of objects of `Student` class.

We need to put them in alphabetical order by name: last name then first name if last names tie.

Q. How do we arrange the items in the array alphabetically by name?

```
public class Student {
   private String fName;
   private String lName;
   private String address;
   private String email;
   private String phone;

   etc....
```

SORT STUDENT SOLUTION- *COMPARABLE*

Problem: An array of `Student` class. Put them in alphabetical order, by last name, first name (if tie).

- Most sorting algorithms use *comparisons* to do the sorting.
- We want to make Student objects comparable to each other.
- Then, we can use any sorting algorithm that works by comparison.

Solution: Use the java library interface Comparable to make Student Objects comparable to each other.

SORT STUDENT SOLUTION- *COMPARABLE*

If Student objects implement the java library interface `Comparable`, then we can use the following sorting methods:

Arrays of objects that implement `Comparable` can be sorted automatically by `Arrays.sort`

ArrayLists of objects that implement Comparable can be sorted automatically by `Collections.sort`

SORT STUDENT SOLUTION- *COMPARABLE*

Problem: An array of `Student` class. Put them in alphabetical order, by name.

Solution: Use the java library interface `Comparable`.

Let's look at *Comparable* before we finish the sorting students problem.

THE COMPARABLE INTERFACE

```
public interface Comparable {

    public int compareTo(Object obj);
```

compareTo(otherComparable) compares a `Comparable` object to `otherComparable`.

- will return **0** if the two `Comparable` objects are **equal.**
- returns a **negative** number if the `Comparable` object **is less** than `otherComparable`
- returns a **positive** number if the `Comparable` object **is greater** than `otherComparable`.

THE COMPARABLE INTERFACE

Intended to model the "natural" ordering of elements in a class – compares 2 objects.

```
public int compareTo(Object other);
```

Return type **int**

Parameter type **Object**

The ***calling*** object: object (`this`) calling the method.
The ***parameter*** object: the single parameter object known as ***other***.

RECALL: THE COMPARABLE INTERFACE

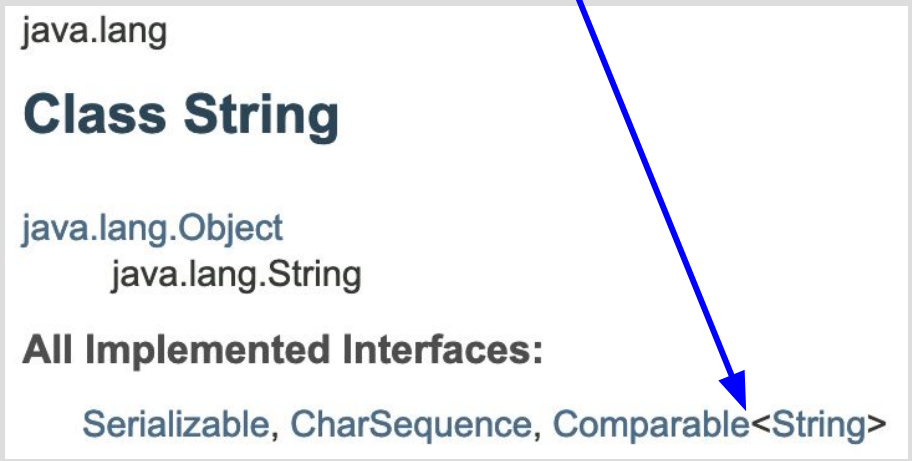
The "meaning" of **compareTo**:
a, b are of some type (cars, or strings, or.....)

`a.compareTo(b)` Ans: < 0
a comes before *b* in natural ordering

`a.compareTo(b)` Ans: == 0
a, b, equal in natural ordering

`a.compareTo(b)` Ans: > 0
a comes after *b* in natural ordering.

Strings are Comparable!



Clicker Question 3

```
public int compareTo (String str1, String str2) {
    return str1.compareTo(str2);
}

//After this method call is executed:
int result = compareTo("Hello", "Hello");
```

Which one of the following is the value referenced by "`result`"?

A. -1
B. 1
C. 0

Clicker Question 3

```
public int compareTo (String str1, String str2) {
    return str1.compareTo(str2);
}

//After this method call is executed:
int result = compareTo("Hello", "Hello");
```

Which one of the following is the value referenced by "`result`"?

A. -1

B. 1

C. 0 CORRECT

Solution: Use the java library interface `Comparable`.

implements Comparable

```
public class Student implements Comparable {

  public int compareTo(Object other){
     Student otherStu = (Student)other;
     int result = this.getLastName().compareTo(otherStu.getLastName());
     // if a tie- look at first name
     if(result==0)
        result = this.getFirstName().compareTo(otherStu.getFirstName());
     return result;
  }
```

Provide an implementation of compareTo

SORT STUDENT SOLUTION- ARRAY USAGE

```
public static void main(String[] args){

  Student stu1 = new Student("Zoe","Smith","44 Harden", "zoes@mymail.com",
                                           "387 833-1234", 2.65, "Finance");
  Student stu2 = new Student("Joe", "Smith", "12 Penny Lane", "joe@geemail.com",
                                            "411 333-1234", 4.00, "Classics");
  Student stu3 = new Student("Moe", "Aasgard","100 Easter Island", "moe@geemail.com",
                                            "777 988-1234", 2.75, "Kinesiology");
  Student stu4 = new Student("Edgar", "Poe", "9 Raven Circle", "poe@geemail.com",
                                            "876 123-4455", 3.50, "Informatics");
  Student stu5 = new Student("Zoe","Smith","12 Sylvan", "zoe@mymail.com",
                                           "222 333-1234", 3.75, "Biology");
```

```
//array sorting
System.out.println("Sorting an array:");
Student[] stuArr = new Student[5];
stuArr[0] = stu1;
stuArr[1] = stu2;
stuArr[2] = stu3;
stuArr[3] = stu4;
stuArr[4] = stu5;

Arrays.sort(stuArr);

for(Student curStudent : stuArr) {
   System.out.println(curStudent.toString());
}
```

Arrays.sort calls the Student compareTo method.

SORT STUDENT SOLUTION- ARRAYLIST USAGE

```
public static void main(String[] args){

  Student stu1 = new Student("Zoe","Smith","44 Harden", "zoes@mymail.com",
                                           "387 833-1234", 2.65, "Finance");
  Student stu2 = new Student("Joe", "Smith", "12 Penny Lane", "joe@geemail.com",
                                            "411 333-1234", 4.00, "Classics");
  Student stu3 = new Student("Moe", "Aasgard","100 Easter Island", "moe@geemail.com",
                                            "777 988-1234", 2.75, "Kinesiology");
  Student stu4 = new Student("Edgar", "Poe", "9 Raven Circle", "poe@geemail.com",
                                            "876 123-4455", 3.50, "Informatics");
  Student stu5 = new Student("Zoe","Smith","12 Sylvan", "zoe@mymail.com",
                                           "222 333-1234", 3.75, "Biology");
```

```
//ArrayList sorting
System.out.println("Sorting an ArrayList:");
ArrayList<Student> arrList = new ArrayList<Student>();
arrList.add(stu1);
arrList.add(stu2);
arrList.add(stu3);
arrList.add(stu4);
arrList.add(stu5);

Collections.sort(arrList);

for(Student curStudent : arrList) {
   System.out.println(curStudent.toString());
}
```

Collections.sort calls the Student compareTo method.

PROBLEM 2

UMass Scheduling wants to

1. compare classroom capacities and print the room location, number, and capacity.
2. check if 2 course offerings have the same location and room number.

Q. How do you compare objects - here with `buildingName`, `roomNumber`, `roomCapacity` attributes?

SOLUTION

Solution:

We could use an array of objects.

Problem: Inefficient with only 2 objects.

Another Solution:

We could just compare the 2 objects.

Use Java's `comparable` interface!

See zyBooks chapter 11.17.

EXAMPLE: IMPLEMENTING THE COMPARABLE INTERFACE

```
public class CourseLocation implements Comparable{
    String buildingName;
    String roomNumber;
    int roomCapacity;

    public CourseLocation(String name, String num, int cap) {
        buildingName = name;
        roomNumber = num;
        roomCapacity = cap;
     }
    public String getBuilding(){
        return buildingName;
     }
    public String getRoomNumber(){
        return roomNumber;
     }
    public int getRoomCapacity(){
        return roomCapacity;
     }
    public String toString(){
        return buildingName+ "   " +roomNumber+ "   " + roomCapacity;
    }
```

`implements` keyword

DEMO

Methods like any regular class

EXAMPLE: IMPLEMENTING THE compareTo METHOD -

Custom criteria in `ClassLocation`

The `compareTo` method returns an integer greater than 0 if the room capacity of the object passed in is less than the capacity of this object, an integer less than 0 if the the room capacity of the object passed in is greater than the capacity of this object, and returns 0 if they have the same room capacity.

```
public int compareTo(Object obj){
    CourseLocation otherLoc = (CourseLocation)obj;
    return roomCapacity-otherLoc.getRoomCapacity();
 }
```

EXAMPLE: IMPLEMENTING THE `isEqualTo` METHOD -

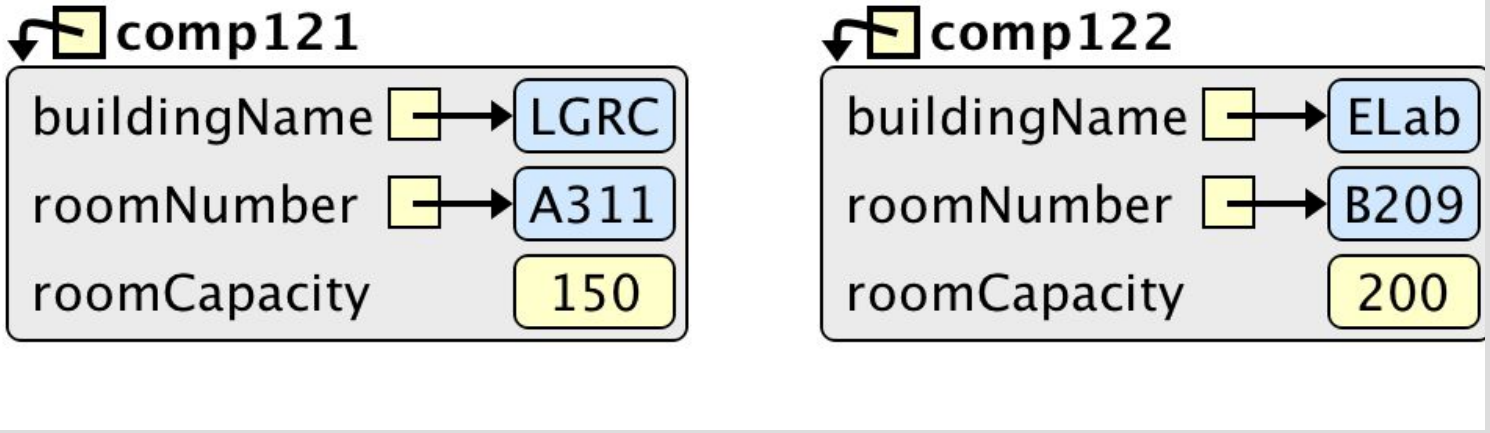
`public boolean equals(Object obj)`
Indicates whether some other object is "equal to" this one.

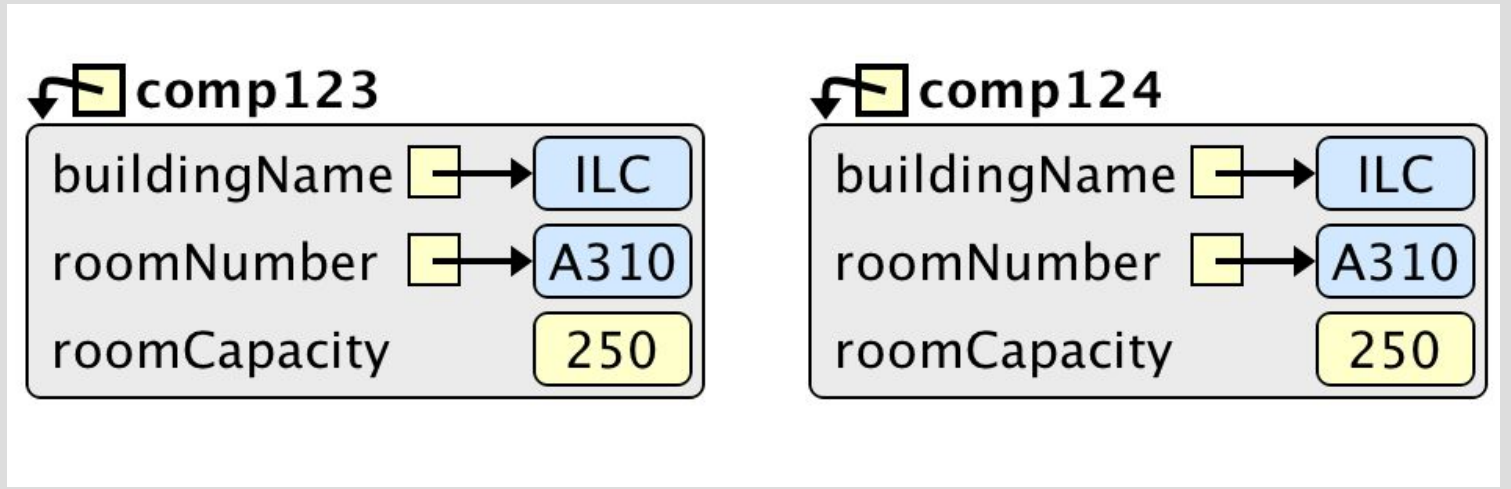
Custom criteria in `ClassLocation`

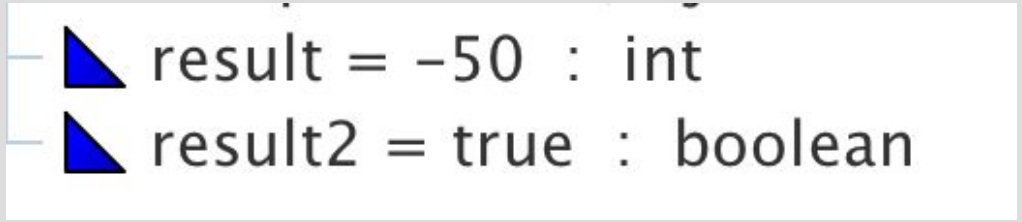
```
public boolean isEqualTo(Object obj){
    CourseLocation otherLoc = (CourseLocation)obj;
    boolean result = false;
    if(buildingName.equals(otherLoc.getBuilding())
            && roomNumber.equals(otherLoc.getRoomNumber()))
        result = true;
    return result;
  }
```

The `isEqualTo` method returns `true` if the object passed in has the same building and room number as this object, `false` otherwise.

EXAMPLE: DEMO OUTPUT








```
-50
true
```

ABSTRACT CLASS vs INTERFACE

Abstract Class

- contains shared data/functionality.
- partial subclass implementation.

Interface

- subclass completely responsible for implementation.

Abstract class: provides "***Template***" for subclasses to use in their specialization of a process. Often, the abstract class offers a partial implementation of a task that the subclasses can specialize.

Interface: provides a guarantee that all implementing classes provide an implementation of a set of public methods. This allows implementing classes to follow their own "***Strategy***" for carrying out a task.

INTERFACES SUMMARY (1)

1. Have no instance variables; **cannot make an interface object.**
2. Constants and method headers are **public by default.**
3. Can have **constants**.
4. Public methods are **disembodied.**
5. Classes **implement** interfaces.
6. Classes can implement **any number of interfaces.**
7. Some interfaces are **library** interfaces; others you **write your own.**

INTERFACES SUMMARY(2)

Consider using interfaces if any of these statements apply to your situation:

1. You expect that unrelated classes would implement your interface.
2. You want to specify the behavior of a particular data type, but not concerned about who implements its behavior.
3. You want to take advantage of multiple inheritance of type.

Also see: Introduction to interfaces

TO DO

- Complete zyBook chapters 10 and 11.
- Start the next project early - good for practising inheritance concepts.
- Visit online office hours for help.
- Post in Moodle or Piazza for help.