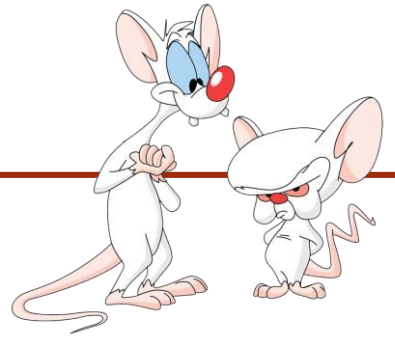
# COMP 250
# INTRODUCTION TO COMPUTER SCIENCE



Week 3.2 : OOD: Comparable

Giulia Alberini, Fall 2020

# WHAT ARE WE GOING TO DO IN THIS VIDEO?



- Java interface Comparable

# COMPARABLE

# JAVA `Comparable` INTERFACE

- The Java `Comparable` interface is used to define an ordering on objects of user-defined class.
- Why would you want that? Well, if you have a list of objects from a given class you might want to be able to sort it.
- `Comparable` is part of `java.lang` package and contains only one method named `compareTo(Object)`.

# JAVA Comparable INTERFACE



```
public interface Comparable<T>{
    int compareTo(T o);
}
```



https://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html

# JAVA `Comparable` INTERFACE

**Some of the methods from certain Java classes use `compareTo()` in their implementation. To function correctly, they assume to be working with `Comparable` generic types. Examples:**



- **`sort()` from `Arrays`.**

**sort**

```
public static void sort(Object[] a)
```

Sorts the specified array of objects into ascending order, according to the natural ordering of its elements. All elements in the array must implement the `Comparable` interface. Furthermore, all elements in the array must be *mutually comparable* (that is, `e1.compareTo(e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the array).

# JAVA `Comparable` INTERFACE

**Some of the methods from certain Java classes use `compareTo()` in their implementation. To function correctly, they assume to be working with `Comparable` generic types. Examples:**



- **`sort()` from `Collections`:**





**sort**

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

Sorts the specified list into ascending order, according to the natural ordering of its elements. All elements in the list must implement the `Comparable` interface. Furthermore, all elements in the list must be *mutually comparable* (that is, `e1.compareTo(e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the list).

# String IMPLEMENTS Comparable

## compareTo

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

Compares two strings lexicographically. The comparison is based on the Unicode value of each character in the strings. The character sequence represented by this `String` object is compared lexicographically to the character sequence represented by the argument string. The result is a negative integer if this `String` object lexicographically precedes the argument string. The result is a positive integer if this `String` object lexicographically follows the argument string. The result is zero if the strings are equal; `compareTo` returns `0` exactly when the `equals(Object)` method would return `true`.



This is the definition of lexicographic ordering. If two strings are different, then either they have different characters at some index that is a valid index for both strings, or their lengths are different, or both. If they have different characters at one or more index positions, let *k* be the smallest such index; then the string whose character at position *k* has the smaller value, as determined by using the < operator, lexicographically precedes the other string. In this case, `compareTo` returns the difference of the two character values at position `k` in the two string -- that is, the value:

```
this.charAt(k)-anotherString.charAt(k)
```

If there is no index position at which they differ, then the shorter string lexicographically precedes the longer string. In this case, `compareTo` returns the difference of the lengths of the strings -- that is, the value:

```
this.length()-anotherString.length()
```

https://docs.oracle.com/javase/7/docs/api/java/lang/String.html

# CLASSES THAT IMPLEMENT `Comparable`

- `Character`, `Integer`, `Float`, `Double`, `BigInteger`, **etc. all implement** `Comparable<T>`.



- **You cannot compare objects of these classes using the "<" operator. Instead use** `compareTo()`.

# HOW TO IMPLEMENT `Comparable`

- **Add `implements Comparable` in the definition of the class.**
- **Implement `compareTo()` inside your class.**



```
public class T implements Comparable<T>{

   public int compareTo(T o) {...}

}
```

# REQUIREMENT FOR IMPLEMENTING `compareTo()`

**Consider two variable `t1` and `t2` or type `T`. Then,**



`t1.compareTo(t2)` **returns**

- negative int , if *t1*<*t2*
- 0 , if *t1*=*t2*
- positive int , if *t1*>*t2*





**The relation should also be anticommutative and transitive.**

**Highly recommended:**

```
(t1.compareTo(t2)==0) == (t1.equals(t2))
```

# EXAMPLE - CIRCLE

- Sometimes deciding how to compare elements of a given type can be straightforward.
- Let's think about the data type Circle.



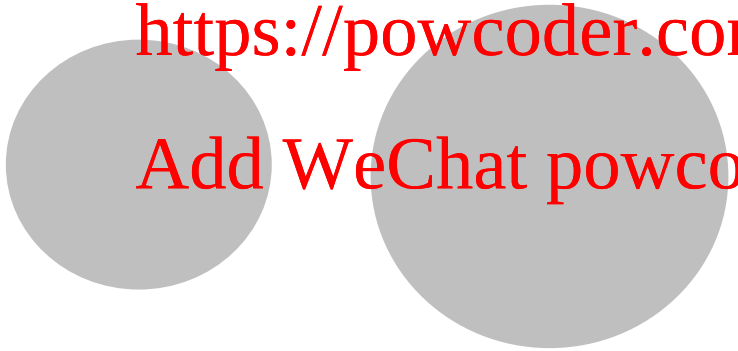
```
public class Circle {
    private double radius;
    :
}
```

- How should we implement `compareTo()` and `equals()` in order to establish a *natural ordering* between elements of type `Circle`?

# EXAMPLE - CIRCLE

- **How should we implement `compareTo()` and `equals()` in order to establish a *natural ordering* between elements of type `Circle`?**





**We could simply compare their radius (or their area).**

# CIRCLE – compareTo()

```
public class Circle extends Shape implements Comparable<Circle>{
   private double radius = 5;

   public int compareTo(Circle c) {
      if (this.radius < c.radius)
         return -1;
      else if (this.radius == c.radius)
         return 0;
      else
         return 1;
   }

   public boolean equals(Object obj) {
      return obj instanceof Circle && ((Circle) obj).radius == this.radius;
   }
}
```

# EXAMPLE – ORC

- Other times, is not so straightforward. Suppose we have created a new data type `Orc`.
- How should we compare and sort elements of this type?



Base on their name? On their height? On their weapon? On who is scarier?

# ORC – compareTo() TAKE 1

```
public class Orc implements Comparable<Orc> {
   private String name;
   private int height;
   private Weapon w;
   public int compareTo(Orc o) {
      if(this.height < o.height) {
         return -1;
      } else if(this.height == o.height) {
         return 0;
      }  else {
         return 1;
      }
   }
}
```



- Note that in this case we probably don't want to consider two Orcs with the same height to be equal.
- This implies that the implementation of compareTo() violates the Java API recommendations.
- Such violation should be clearly indicated using the following language: "Note: this class has a natural ordering that is inconsistent with equals."

# ORC – compareTo() TAKE 2

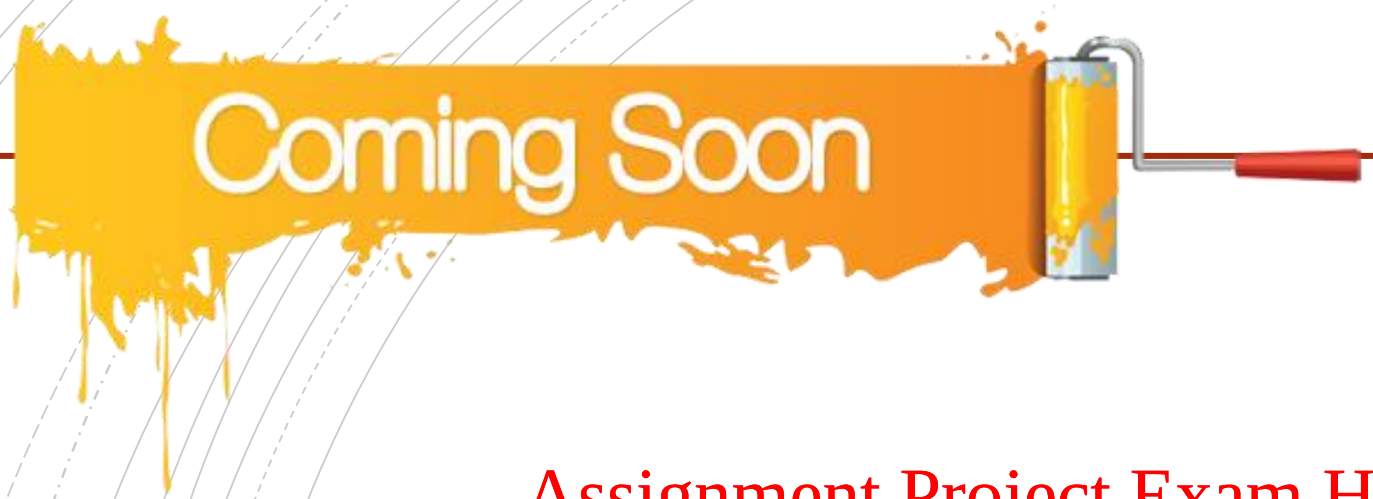
```
public class Orc implements Comparable<Orc> {
    private String name;
    private Integer height;
    private Weapon w;
    public int compareTo(Orc o) {
        int result = this.w.compareTo(o.w);
        if(result==0) {
            result = this.height.compareTo(o.height);
        }
        if(result == 0) {
            result = this.name.compareTo(o.name);
        }
        return result;
    }
}
```



- We can also use compareTo() to compare multiple characteristics.
- Generally, it is better to reuse existing code than to write our own. Thus, in this case, we can use the *compareTo()* methods from other classes to.

# TO RECAP

- `Comparable` **defines a natural ordering.**
- **If you define a new data type for which sorting makes sense to you, then you should implement** `Comparable` **to define a natural ordering on objects of such type.**






In the next videos:



- Iterable and Iterator