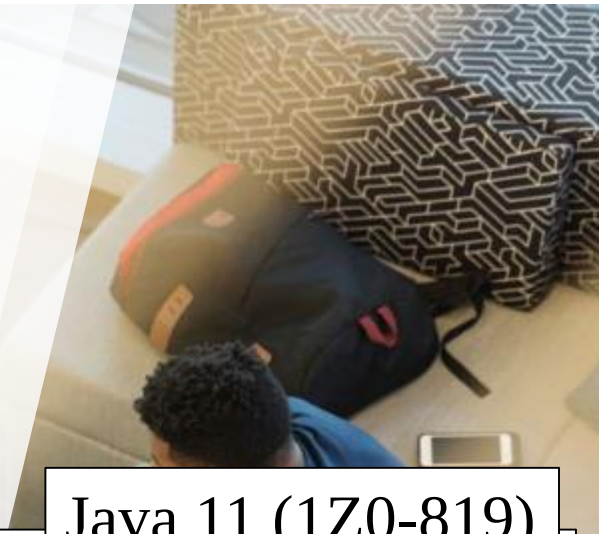
Collections

Sorting – Comparable and Comparator

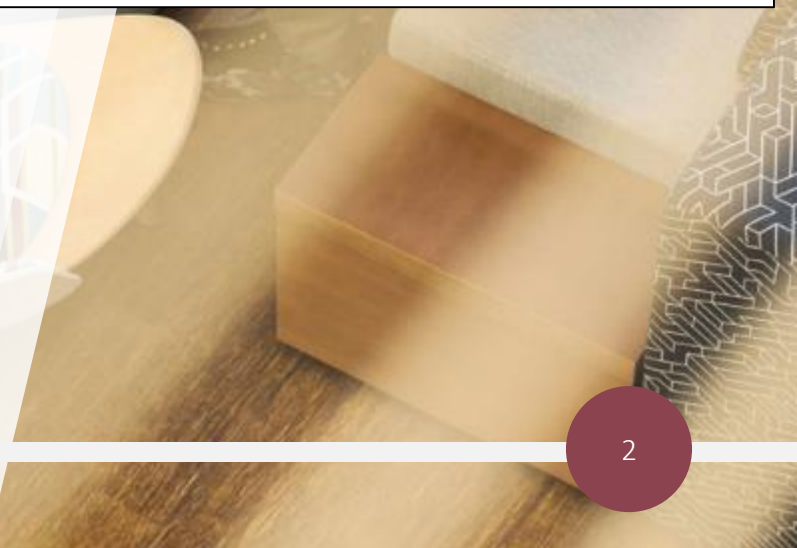
Collections



Java 11 (1Z0-819)

Working with Arrays and Collections

- ✓ Use generics, including wildcards
- ✓ Use a Java array and List, Set, Map and Deque collections, including convenience methods
- → Sort collections and arrays using Comparator and Comparable interfaces

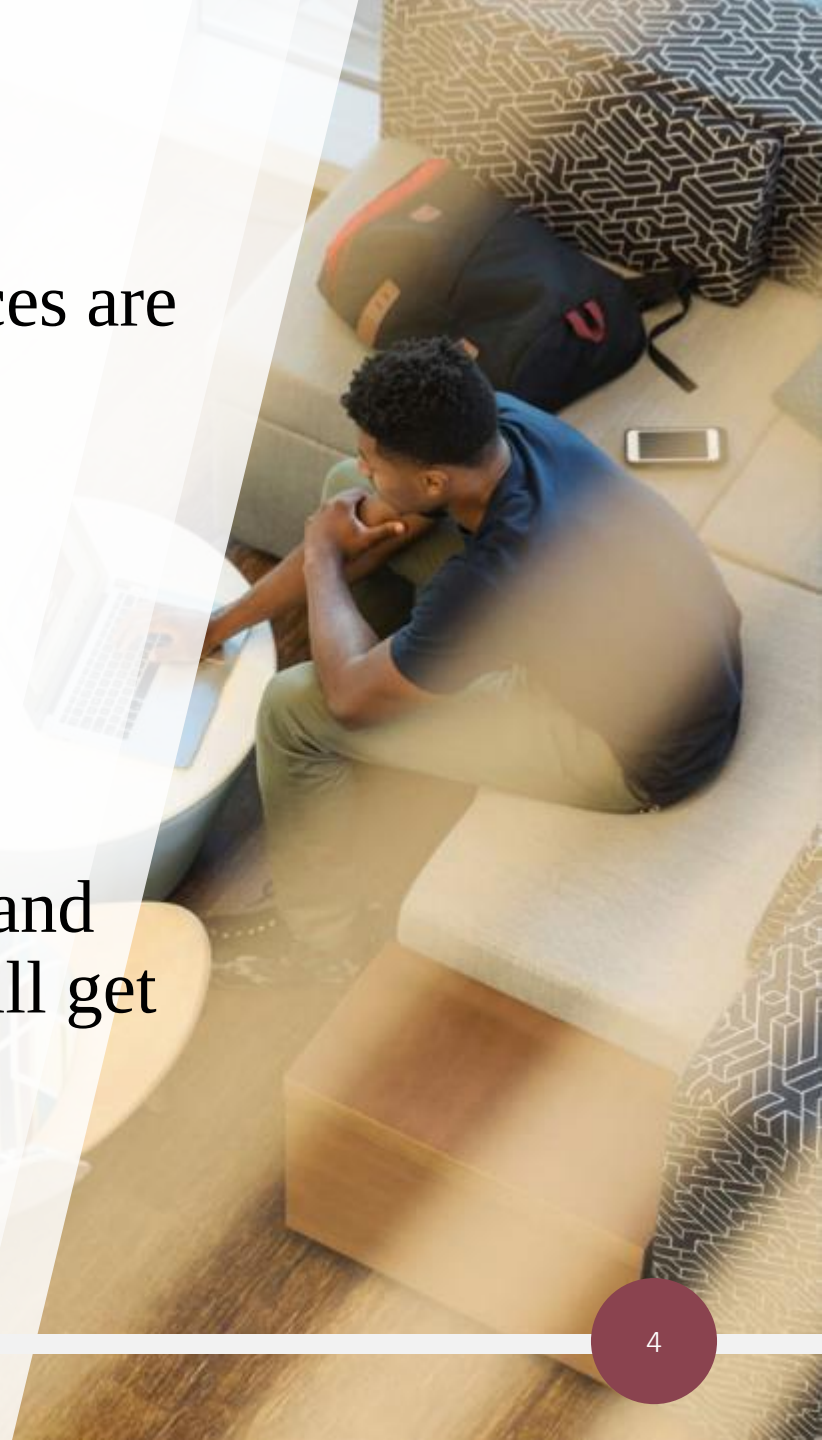


Sorting

- Both collections and arrays can be sorted and searched using methods in the API.
- The *Collections* class is a utility class i.e. a class which consists exclusively of static methods, used for operating on collections.
- The *Arrays* class is also a utility class; the *Arrays* class however, operates on native arrays only ([] syntax).
- One can convert an array (of reference types) to a *List* using the *Arrays.asList* method. The returned *List* can then be passed to useful methods that exist in the *Collections* class.

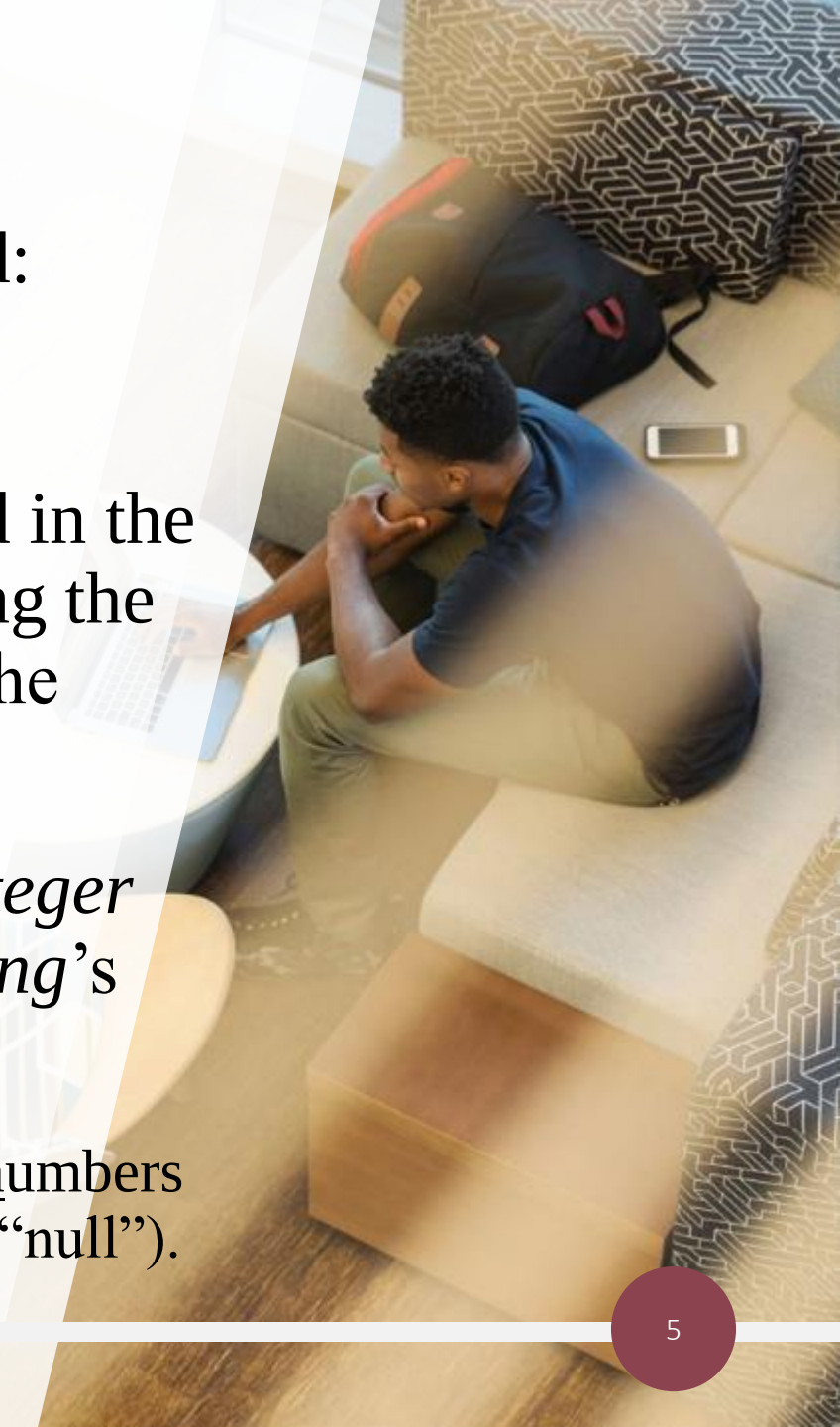
Comparable and *Comparator* interfaces

- The *Comparable<T>* and *Comparator<T>* interfaces are used for comparing objects of similar type.
- Both are functional interfaces.
- Sorting is a classic example where they are used.
- *java.lang.Comparable* and *java.util.Comparator*
- Note: if you add an object of a class to e.g. *TreeSet* and the class does NOT implement *Comparable*, you will get a *ClassCastException*.



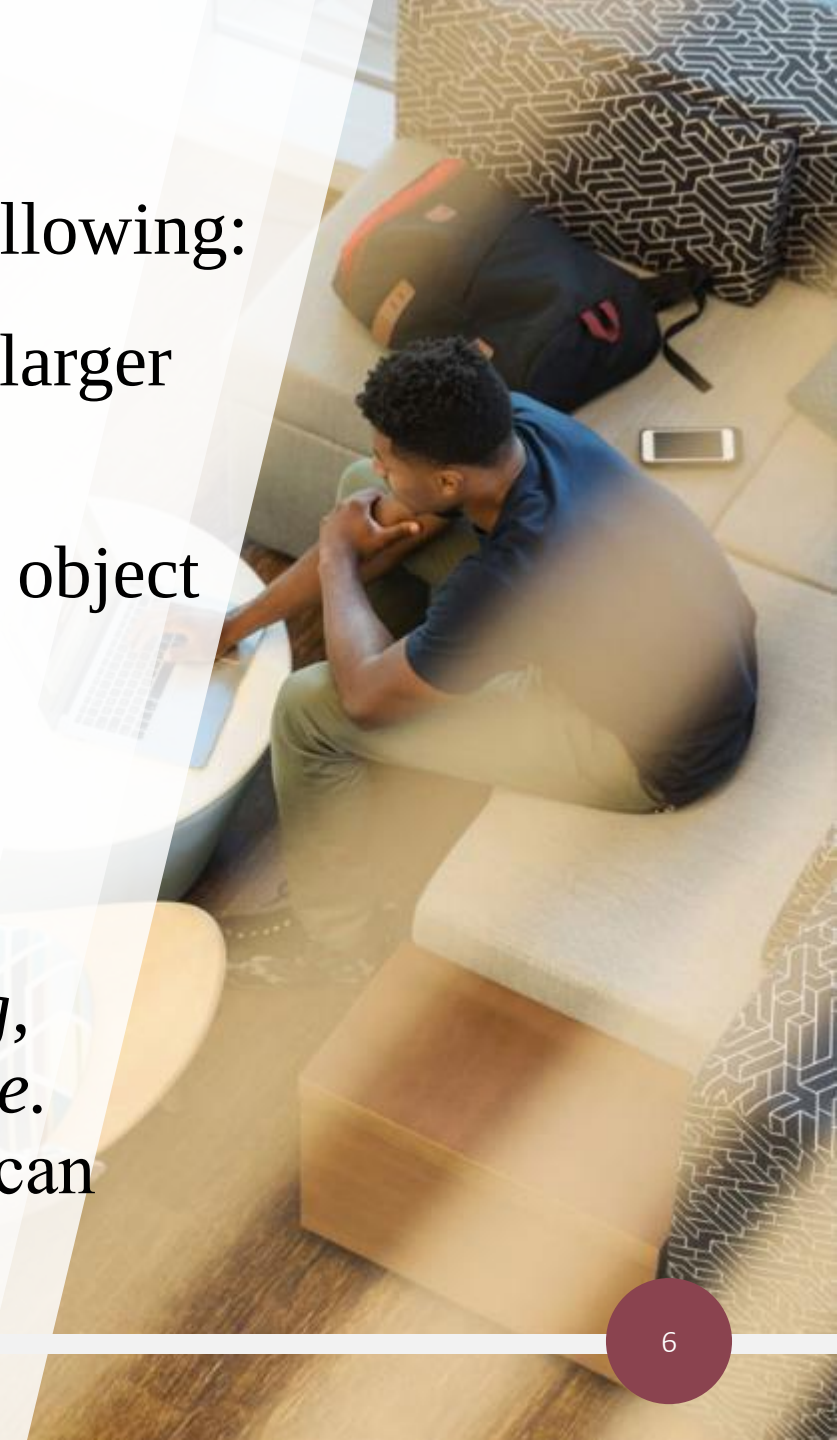
Comparable interface

- The *Comparable<T>* interface defines one method:
  - *int compareTo(T that)*
- Given that you implement the *compareTo()* method in the class itself, you already have access to its state using the "*this*" reference. Thus, you can compare "*this*" to the object passed in ("*that*" above).
- *Comparable* defines the "natural ordering". For *Integer* this ascending numeric order (1, 2, 3 etc..); for *String*'s it is alphabetic order ("A", "B", "C" etc..).
  - Note: *TreeSet* would sort *String*s according to Unicode: numbers before letters, uppercase letters before lowercase letters ("null").



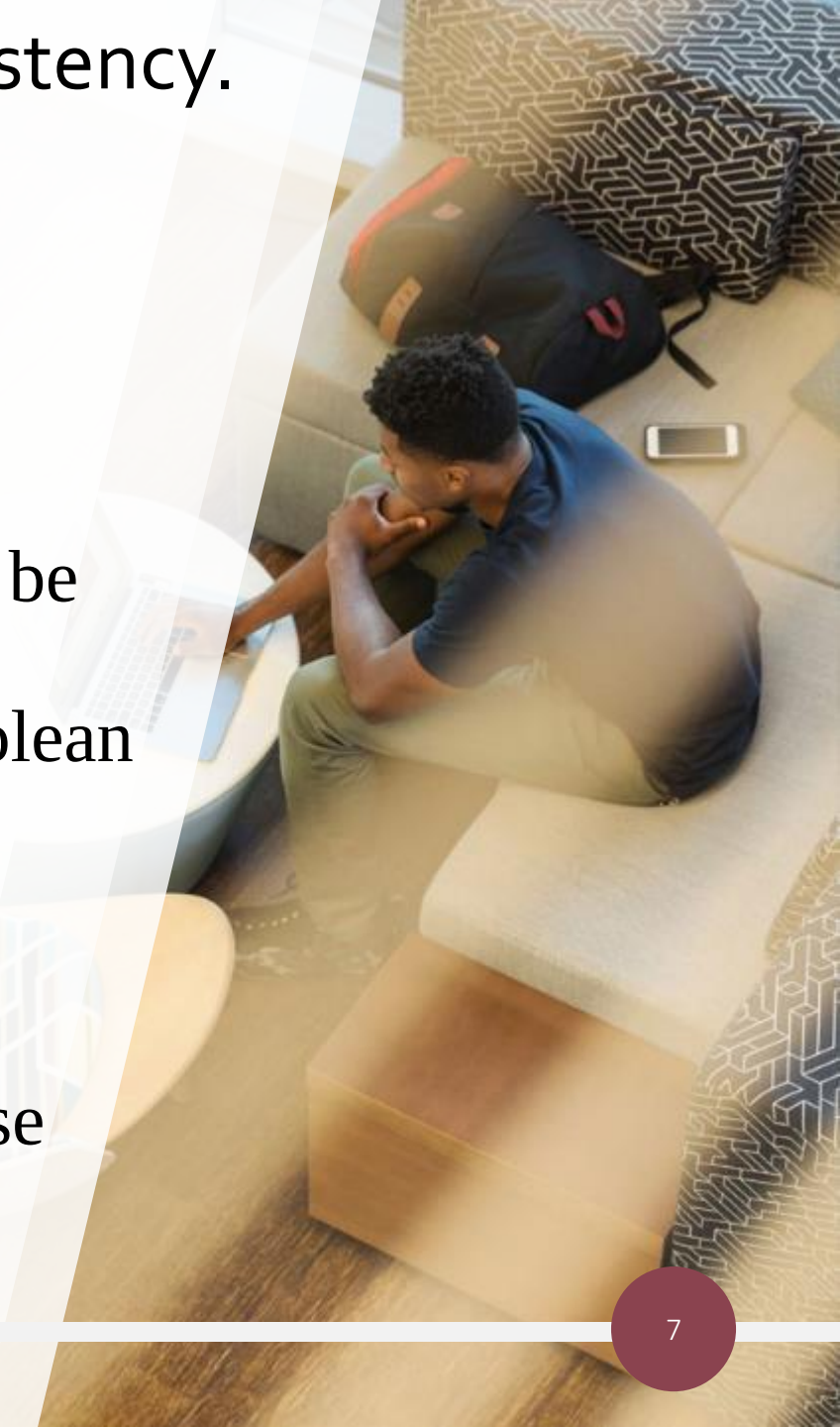
Comparable interface

- *compareTo* logic : return an *int* value based on the following:
  - return a positive number if the current object is larger than the object passed in
  - return 0 if the current object is equivalent to the object passed in
  - return a negative number if the current object is smaller than the object passed in
- This logic can be delegated to existing types (*String*, *Integer*) that already have implemented *Comparable*. In other words, if you are comparing *Integer*'s you can delegate.



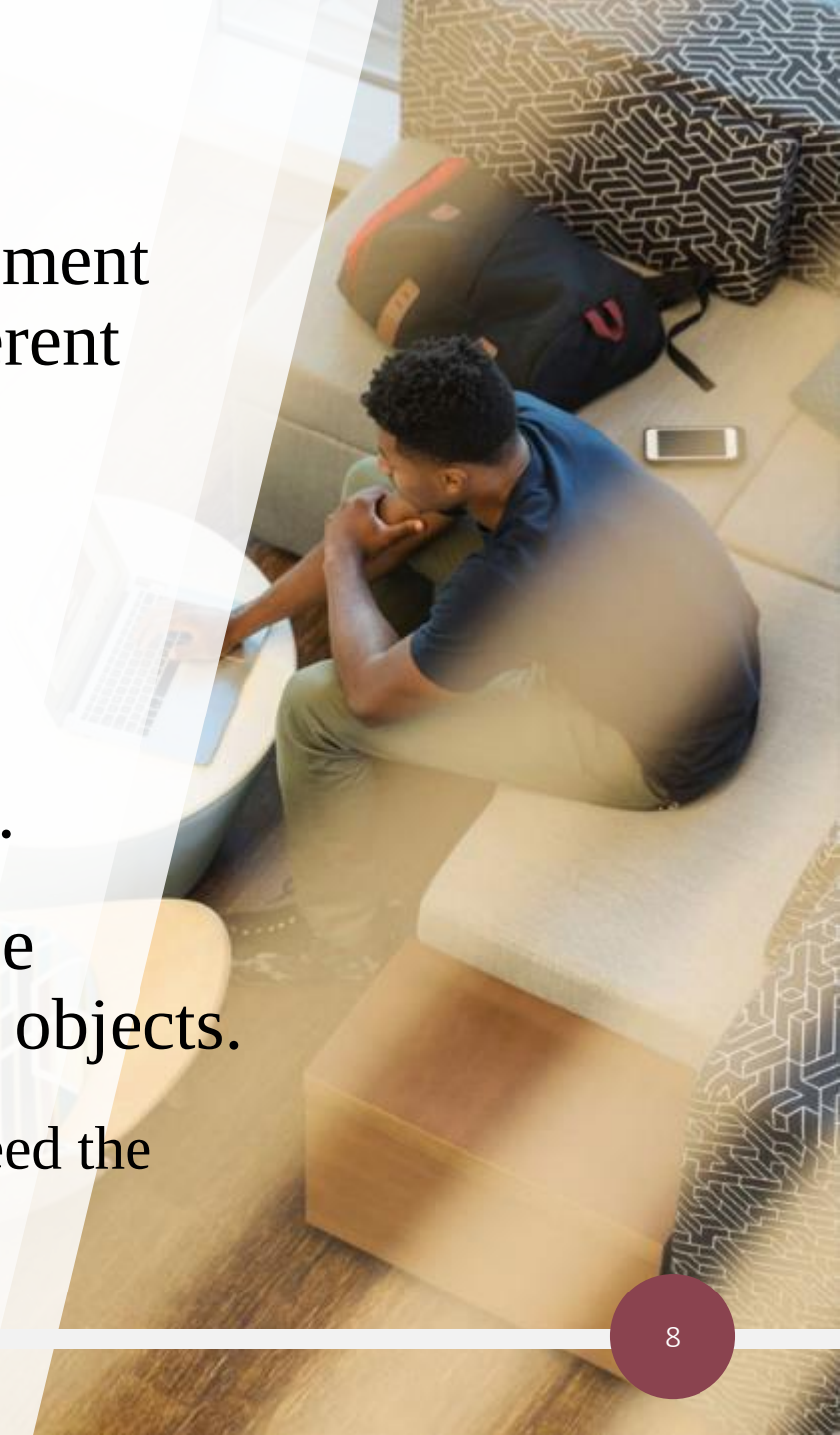
Comparable - *compareTo()* and *equals()* consistency.

- When are 2 objects equal?
  - *compareTo()* – returns 0
  - *equals()* – returns *true*
- API: “The natural ordering for a class `C` is said to be *consistent with equals* if and only if `e1.compareTo(e2) == 0` has the same boolean value as `e1.equals(e2)` for every `e1` and `e2` of class `C`”.
- We are “strongly recommended” to keep our *Comparable* classes consistent with equals because “sorted sets (or sorted maps)…behave strangely” otherwise. [API]



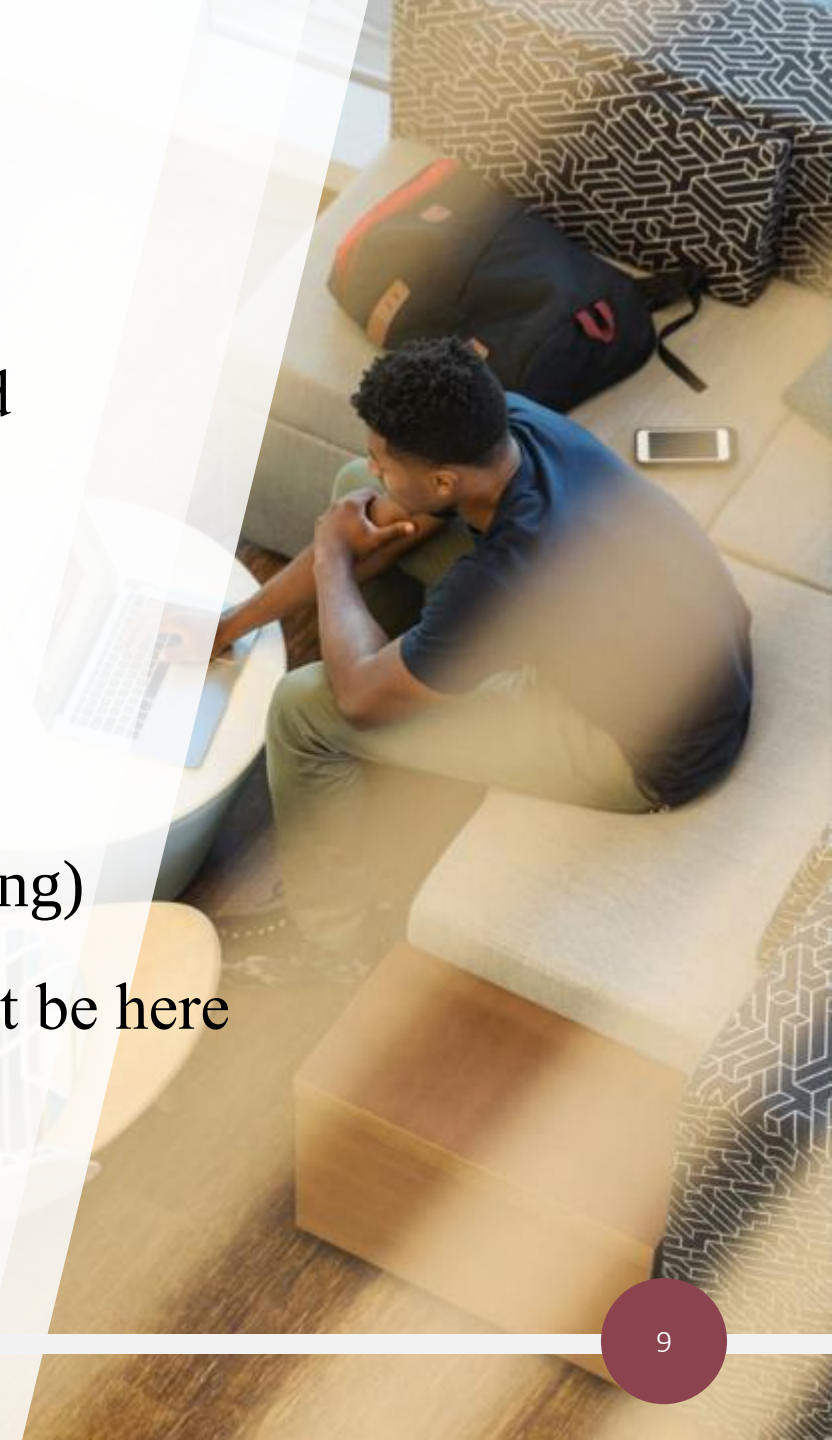
Comparator interface

- What if the objects we wanted to sort did not implement *Comparable* or if we wanted to sort in several different ways? Answer: *Comparator*.
- *Comparator* is also a functional interface:
  - *int compare(T o1, T o2)*
- The logic internally is the same as for *compareTo()*.
- Typically, this is coded externally to the class whose objects we are comparing so we need to compare 2 objects.
  - as *Comparable* is coded internally to the class, we just need the one/other object we want to compare to '*this*' object



How to remember the differences?

- Compara*t*or - "**T**wo out of three ain't bad"
  - "Compara**t**or takes **t**wo" args but not the "**T**o" method
    - int compare(T o1, T o2)
    - ORE = Comparat**or** and compar**e**()
- Comparable
  - if *Comparator* takes 2 then this takes 1 (natural ordering)
  - as *Comparator* does not have the "To" method, it must be here
    - int compareTo(T o)
    - LEO = Comparab**le** and compareT**o**()



binarySearch()

- *binarySearch()* requires a sorted *List*.
- As with *sort()*, if you don't want natural order, you can pass in a comparator.