Announcements
• Project 2 due 11/01

Material
• Searching (linear, binary)
• Using Interfaces: Polymorphism
  • Comparable and Comparator
• Linear search - modified to use Comparator
• Anonymous classes
Searching

• Looking for an item in a collection
• For linear collections

Two main approaches
Linear search (brute force approach)
Binary search
Linear Search

• Brute force:

```java
boolean linearSearch(Object[] data, Object target) {
    int index = 0;
    boolean found = false;

    while (!found && index < data.length) {
        if (data[index].equals(target)) {
            found = true; // return index
            index++;
        }
    }

    return found; // return -1
}
```

Let's explore this method

case 1: search for Integers

```
ray = [56, 23, 66, 44, 78]
start
```

Let's explore this method
case 1: search for Integers
What if...

- Consider class
  ```java
class Student {
    private String name;
    private String id;

    // compares both name and id
    public boolean equals(Student s) {
      // ... }
  }
```

- Consider an array of these...
  ```java
  Student[] registrar = ...
  Student target = ...
  linearSearch (registrar, target);
  ```

- What if I want to find the student by name, because I don’t know the ID, or the other way around?
Comparator Interface

• Compare two objects

```java
interface Comparator {
    int compare(Object o1, Object o2)
    ...
}
```

• Flexible Comparisons
  • Compares two object parameters (unlike `compareTo()` which compares explicit to implicit parameter)
  • An object of a class implementing `Comparator` & providing only a `compare()` method is termed a *function object*

Let's explore this method case 3: modify linear search
• Now modify `linearSearch` to take a comparator object

```java
boolean linearSearch
        (Object[] data, Object target, Comparator comp)
{
    ...
    if (comp.compare(data[index], target) == 0)
        found = true;
    ...
}
```

Let's explore this method

Case 3: modify linear search
• Define a comparator for Name
  
  class CompareByName implements Comparator {
    int compare(Student o1, Student o2) {
      String name1 = o1.getName();
      String name2 = o2.getName();
      return name1.compareTo(name2);
    }
  }

• Define a comparator for ID
  
  class CompareByID implements Comparator {
    int compare(Student o1, Student o2) {
      String id1 = o1.getID();
      String id2 = o2.getID();
      return id1.compareTo(id2);
    }
  }
What did we just do?

• Separated algorithm from the parts that are external to it
• Used an interface to specify what external pieces are missing
• This is excellent use of OO techniques
  • Called polymorphism (using generic code)
Polymorphism

• Definition
  • Poly = many, Morph = forms
  • Polymorphism = many forms

• In OO refers to the ability to have a single name represent many different entities, that is for a single name to take many forms

• Often used with inheritance (e.g. toString() is different in every class), but also applies to interfaces
• The loop to do the search is very generic, can we extract it out?

```java
public interface SearchType {
    /**
     * Returns the prompt to be used in the main loop
     * @return string prompt to be used
     */
    public String getPrompt();

    /**
     * Routine intended to do the searching, it returns true if the item sought is found. The input is the string that the user entered on the keyboard.
     * @param input string entered by the user on the keyboard
     * @return true if the item is found, false otherwise
     */
    public boolean doSearch(String input);
}
```