Inheritance With Comparators

Side Note

- Methods can take generic types
- This method compares two LinkedListNodes of any type
- T will be replaced with the type of the lists given as arguments

```
public <T> void compareLinkedLists(LinkedListNode<T> l1, LinkedListNode<T> l2) {
    if(!(l1 == null && l2 == null)){
        assertTrue("l1 was null, but l2 was not", l1 != null);
        assertTrue("l2 was null, but l1 was not", l2 != null);
        assertEquals("node values are not equal",l1.getValue(),l2.getValue());
        compareLinkedLists(l1.getNext(), l2.getNext());
    }
}
```

Comparators

Comparators

The Problem:

- Need to compare 2 values
- Determine which of the 2 values comes first in sorted order
- Use the comparator to sort any number of values

The Solution:

- Use a comparator

Comparators

- A comparator takes 2 parameters and returns information about the order of the two inputs
- Comparators come it 2 primary styles:
  - Returns an int
    - Negative if the first input comes before the second
    - Positive if the second input comes before the first
    - 0 if the order is tied
  - Returns a boolean
    - True if the first input comes before the second
    - False otherwise - Including ties

Comparators

Returns an int

- Many Java classes contain a compareTo method
- This method returns an int depending on the order of the inputs
  - If the calling object (this) comes before the argument -> a negative int
  - If the calling object (this) comes after the argument -> a positive int
  - If the the order is tied -> 0
- Common for the ints to be -1 and 1, but not guaranteed in all cases

```
String one = "a";
String two = "b";
System.out.println(one.compareTo(two));
System.out.println(one.compareTo(one));
System.out.println(two.compareTo(one));
```

-1
0
1

Comparators

- For Strings, comparisons are made alphabetically
  - Compare each character of both Strings starting with the first
    - If the characters are different, the String with the character that comes first in the alphabet is determined to come first in the ordering
    - If the characters are the same, check the next characters
    - If the end of one String is reached, that String comes first
- compareTo checks the case of each character!
  - All lowercase letter come after all capital letters ("Z" comes before "a")!

```
String one = "a";
String two = "b";
System.out.println(one.compareTo(two));
System.out.println(one.compareTo(one));
System.out.println(two.compareTo(one));
```

-1
0
1

Comparators

Returns a boolean

- Our example in class will return a boolean

- Take 2 parameters
- Return true if the first parameter comes first
- Return false otherwise - including ties

Writing Comparators

Comparators

- Our example in class will return a boolean

- This method will compare ints in decreasing order
  - return true when a > b (a comes before b)
  - false otherwise - including a == b

```
public boolean compare(int a, int b) {
    return a > b;
}
```

Comparators

- We'll wrap this method in a class
- Note that the compare method is not static
  - Need to create an object of this type to call the method

- Ok.. hey, wait.. there's no constructor! How we gonna create an object?

```
public class IntDecreasing {
  public boolean compare(int a, int b) {
    return a > b;
  }
}
```

Default Constructor

- If you don't write a constructor, you automatically get the default constructor
  - The default constructor takes no parameters and has no code in it's body
- These two classes are functionally identical

```
public class IntDecreasing {

  public IntDecreasing(){}

  public boolean compare(int a, int b) {
    return a > b;
  }
}
```

```
public class IntDecreasing {
  public boolean compare(int a, int b) {
    return a > b;
  }
}
```

Comparators

- And we'll use inheritance!

- Write a Comparator class that takes a generic type
- Extend this class with a specific type
- Since we're switching to generics, we change int to Integer

```
public class Comparator<T> {
  public boolean compare(T a, T b) {
    return false;
  }
}
```

```
public class IntDecreasing extends Comparator<Integer> {
  @Override
  public boolean compare(Integer a, Integer b) {
    return a > b;
  }
}
```

Comparators

- The comparator class defines a stubbed out compare method that always returns false

- Our IntDecreasing class overrides compare

```
public class Comparator<T> {
  public boolean compare(T a, T b) {
    return false;
  }
}
```

```
public class IntDecreasing extends Comparator<Integer> {
  @Override
  public boolean compare(Integer a, Integer b) {
    return a > b;
  }
}
```

Comparators

- We can extend this comparator to compare any type in any way
- This comparator compares GameItems based on their distance from (0, 0)
- We'll stick with the int comparator for lecture to keep things simpler

```
public class Comparator<T> {
  public boolean compare(T a, T b) {
    return false;
  }
}
```

```
public class DistanceFromOrigin extends Comparator<GameItem> {

    private double distance(GameItem item){
        return Math.sqrt(Math.pow(item.getX(), 2.0) + Math.pow(item.getY(), 2.0));
    }

    @Override
    public boolean compare(GameItem a, GameItem b) {
        return distance(a) < distance(b);
    }
}
```

Sorting With Comparators

- We have comparators now, but how do we use them?
- We want to sort any number of values
  - How do we do this when we can only compare 2 values?

```
public class IntDecreasing extends Comparator<Integer> {
  @Override
  public boolean compare(Integer a, Integer b) {
    return a > b;
  }
}
```

```
public class DistanceFromOrigin extends Comparator<GameItem> {

    private double distance(GameItem item){
        return Math.sqrt(Math.pow(item.getX(), 2.0) + Math.pow(item.getY(), 2.0));
    }

    @Override
    public boolean compare(GameItem a, GameItem b) {
        return distance(a) < distance(b);
    }
}
```

Insertion Sort

- Insertion sort:
  - For each value in the list to be sorted
    - Find where that value belongs in the output list
    - Insert the value at the location

input: **[1 6 5]**

output: **[ ]**

Insertion Sort

- The first element is copied to the output list
- No decision to make since the output is empty

input: [1 6 5]

output: [1]

Insertion Sort

- Find where 6 is inserted
- Compare 1 and 6 using our comparator
  - The comparator returns false since 1 does not come before 6 (Remember we're sorting in reverse order)
- Insert 6 before 1

input: **[1 6 5]**

output: **[6 1]**

Insertion Sort

- Find where 5 is inserted
- Compare 6 and 5 using our comparator
  - The comparator returns true so we advance to the next element
- Compare 1 and 5 using our comparator
  - The comparator return false. This is where we insert

input: **[1 6 5]**

output: **[6 5 1]**

Sorting With Comparators

```
public class Sort<T> {
  Comparator<T> comparator;
  public Sort(Comparator<T> comparator){
    this.comparator = comparator;
  }
  public ArrayList<T> sort(ArrayList<T> input) {
    ArrayList<T> output = new ArrayList<>();
    for (T valueToInsert : input) {
      int location = 0;
      for (T valueToCompare : output) {
        if (comparator.compare(valueToCompare, valueToInsert)) {
          location++;
        }
      }
      output.add(location, valueToInsert);
    }
    return output;
  }
  public static void main(String[] args) {
    Sort<Integer> sorter = new Sort<>(new IntDecreasing());
    ArrayList<Integer> list = new ArrayList<>(Arrays.asList(1, 6, 5));
    ArrayList<Integer> output = sorter.sort(list);
    System.out.println(output);
  }
}
```

- Let's look through the code for insertion sort using our comparator

Sorting With Comparators

```
public class Sort<T> {
  Comparator<T> comparator;
  public Sort(Comparator<T> comparator){
    this.comparator = comparator;
  }
  public ArrayList<T> sort(ArrayList<T> input) {
    ArrayList<T> output = new ArrayList<>();
    for (T valueToInsert : input) {
      int location = 0;
      for (T valueToCompare : output) {
        if (comparator.compare(valueToCompare, valueToInsert)) {
          location++;
        }
      }
      output.add(location, valueToInsert);
    }
    return output;
  }
  public static void main(String[] args) {
    Sort<Integer> sorter = new Sort<>(new IntDecreasing());
    ArrayList<Integer> list = new ArrayList<>(Arrays.asList(1, 6, 5));
    ArrayList<Integer> output = sorter.sort(list);
    System.out.println(output);
  }
}
```

- The Sort class will take a generic type
  - We write sorting code 1 time ever and use it to sort any type
- The Sort constructor takes and stores a Comparator
  - We write sorting code 1 time ever and use it to sort type in any order

Sorting With Comparators

```
public class Sort<T> {
  Comparator<T> comparator;
  public Sort(Comparator<T> comparator){
    this.comparator = comparator;
  }
  public ArrayList<T> sort(ArrayList<T> input) {
    ArrayList<T> output = new ArrayList<>();
    for (T valueToInsert : input) {
      int location = 0;
      for (T valueToCompare : output) {
        if (comparator.compare(valueToCompare, valueToInsert)) {
          location++;
        }
      }
      output.add(location, valueToInsert);
    }
    return output;
  }
  public static void main(String[] args) {
    Sort<Integer> sorter = new Sort<>(new IntDecreasing());
    ArrayList<Integer> list = new ArrayList<>(Arrays.asList(1, 6, 5));
    ArrayList<Integer> output = sorter.sort(list);
    System.out.println(output);
  }
}
```

- Write a sort method that will take an ArrayList as an input
- Return a new ArrayList with the same values, but in sorted order

- The input is passed by reference
  - Any change made to the input ArrayList will change the state of the heap
  - This change will be seen outside your method
- Must be careful when handling parameters that are references

Sorting With Comparators

```
public class Sort<T> {
  Comparator<T> comparator;
  public Sort(Comparator<T> comparator){
    this.comparator = comparator;
  }
  public ArrayList<T> sort(ArrayList<T> input) {
    ArrayList<T> output = new ArrayList<>();
    for (T valueToInsert : input) {
      int location = 0;
      for (T valueToCompare : output) {
        if (comparator.compare(valueToCompare, valueToInsert)) {
          location++;
        }
      }
      output.add(location, valueToInsert);
    }
    return output;
  }
  public static void main(String[] args) {
    Sort<Integer> sorter = new Sort<>(new IntDecreasing());
    ArrayList<Integer> list = new ArrayList<>(Arrays.asList(1, 6, 5));
    ArrayList<Integer> output = sorter.sort(list);
    System.out.println(output);
  }
}
```

- For each element in the input ArrayList:
  - Find the index in the output where it should be inserted
- Call the comparator to compare each value in the output with the value being inserted
- Count the number of times compare returned true
  - This is the number of elements that come before this value
  - This is the index where the value should be inserted

Sorting With Comparators

```
public class Sort<T> {
  Comparator<T> comparator;
  public Sort(Comparator<T> comparator){
    this.comparator = comparator;
  }
  public ArrayList<T> sort(ArrayList<T> input) {
    ArrayList<T> output = new ArrayList<>();
    for (T valueToInsert : input) {
      int location = 0;
      for (T valueToCompare : output) {
        if (comparator.compare(valueToCompare, valueToInsert)) {
          location++;
        }
      }
      output.add(location, valueToInsert);
    }
    return output;
  }
  public static void main(String[] args) {
    Sort<Integer> sorter = new Sort<>(new IntDecreasing());
    ArrayList<Integer> list = new ArrayList<>(Arrays.asList(1, 6, 5));
    ArrayList<Integer> output = sorter.sort(list);
    System.out.println(output);
  }
}
```

- To create an object of type Sort:
  - Call the constructor with a Comparator as the argument

```
new Sort<>(new IntDecreasing())
```

- Wait. What?
  - IntDecreasing != Comparator
  - Is that allowed?

Sorting With Comparators

```
public class Sort<T> {
  Comparator<T> comparator;
  public Sort(Comparator<T> comparator){
    this.comparator = comparator;
  }
  public ArrayList<T> sort(ArrayList<T> input) {
    ArrayList<T> output = new ArrayList<>();
    for (T valueToInsert : input) {
      int location = 0;
      for (T valueToCompare : output) {
        if (comparator.compare(valueToCompare, valueToInsert)) {
          location++;
        }
      }
      output.add(location, valueToInsert);
    }
    return output;
  }
  public static void main(String[] args) {
    Sort<Integer> sorter = new Sort<>(new IntDecreasing());
    ArrayList<Integer> list = new ArrayList<>(Arrays.asList(1, 6, 5));
    ArrayList<Integer> output = sorter.sort(list);
    System.out.println(output);
  }
}
```

- IntDecreasing != Comparator
- Is that allowed?

- Yes. Yes it is!
- It's allowed because IntDecreasing *extends* Comparator
- **This is polymorphism**
  - Much more discussion about this later in the course
  - For now, this is allowed and we'll use it to sort using any type that extends Comparator