Introduction to Java CS9053
Spring 2024
Prof. Dean Christakos
Midterm
Due: March 8, 2024 11:59
NO EXTENSIONS
NO COLLABORATION

Part I: Inheritance

Here is a list of classes. Classes in bold are abstract.

1) Construct a class hierarchy from the classes and implement them in Java with getters and setters, `equals` methods that compare attributes, `toString` methods, and unique ids. (don't use ids for "`equals`" comparisons)

SportsPlayer（总）- (int)weight, (未知)gender
ShotputPlayer（铅球）- (int)maxDistance（单位 cm)
TrackPlayer（跑道）- (int)distance（单位 m)
BaseballPlayer（棒球）- (int)rbi
BallSportsPlayer（球类）
BasketballPlayer（篮球）- (int)height（单位 cm)
VolleyballPlayer（排球）- (int)maxPoints
PoleVaultPlayer（撑杆跳）- (int)maxHeight（单位 cm)
CrossCountryPlayer（越野跑）- (double)bestMileTime
FieldSportsPLlayer（田赛）
RunningSportsPlayer（径赛）

- All SportsPlayer objects have an integer weight and a gender attribute that is male or female.
- Runners are either in Cross Country or Track.
- Field sports include Shotput and Pole Vault
- Basketball players have a listed height in integer centimeters
- TrackPlayers have a distance attribute in integer meters
- BaseballPlayers have an integer "rbi" attribute
- ShotputPlayers have a "maxDistance" attribute that is an integer centimeters
- PoleVaultPlayers have a "maxHeight" attribute in integer centimeters
- VolleyballPlayers have an integer "maxPoints" attribute
- CrossCountryPlayers have a double "bestMileTime" attribute

2) `SportsPlayer` objects should implement the `Comparable` interface and be `Comparable` based on weight.

3) Create 2 of each concrete player objects. Place them in an `ArrayList` that can only contain `SportsPlayer` objects or their subclasses.

4) Sort that `ArrayList` by weight in ascending order and print out the results. Then sort in descending order and print out the results

5) Implement `getAverageWeight` which takes an `ArrayList` of `SportsPlayers` or an `ArrayList` of any subclass of `SportsPlayers` and calculates and returns the average weight of the elements of the `ArrayList`

Part II

Here you are going to implement a key-value `Pair` object and put in a binary search tree, where you will be able to (optionally) specify the search method for the binary search tree.

1) The `Pair` object:

   A pair object has two fields, a Key and a Value
   They can be retrieved by `getKey` and `getValue` and set with `setKey` and `setValue`
   Key and Value can also be set with the constructor
   Key and Value can be of any types, but these types must be set at compile time by parameterization:

   So, a Pair with `Integer` Keys and `String` Values is created like this:

   ```
   Pair<Integer, String> p = new Pair<Integer, String>(50, "Bob");
   ```

   The `toString` method should return:

   ```
   Pair [key=<key>, value=<value>]
   ```

   Where `<key>` and `<value>` are the `toString` representations of the Key and Value values of the `Pair` object

2) Next you're going to implement the `BinarySearchTree`. The `BinarySearchTree` is made up of `TreeNode` objects

   The `TreeNode` class should be able to be parameterized with any value, including `Pair`.

   The `BinarySearchTree` should use the natural ordering of the value types of the `TreeNode`, unless a comparator is set.

   For example, I could create a `TreeNode` for String objects:

   ```
   TreeNode<Strong> tn = new TreenNode<String>("Hello");
   ```

   This `BinarySearchTree` of `String`s will use the natural ordering of `String`s:

   ```
   BinarySearchTree<String> bst = new BinarySearchTree<String>();
   ```

   However, this `BinarySearchTree` will order the nodes in order of `String` length:

   ```
   BinarySearchTree<String> bst = new BinarySearchTree<String>((s1,
   s2) -> return s1.length() - s2.length());
   ```

Finally, you will create a `BinarySearchTree` of `Pair` objects, where **the nodes are ordered by the `Key` value of the `Pair` value** in the `TreeNode`.

You don't have to implement the insert/search methods for a `BinarySearchTree`. I'm not a monster. You do, however, have to make it compatible with parameterizable `TreeNode` objects, allow use the `TreeNode` values' natural ordering for insert and search or allow a `Comparator` to be set that will use an ordering you pass into in the `BinarySearchTree` constructor.

So, for example, I should be able to create a `Pair` objects that are parameterized with `Integer` `Key`s and `String` `Value`s.

Then I should be able to create a `BinarySearchTree` that takes `Pair` objects, insert `Pair` objects, and show the Pair objects in the `BinarySearchTree` in ascending order of `Key` value. Alternately, I should be able to create a `BinarySearchTree` of Pair objects in descending order of `Key` value, or ordered by the `Value` field.

Hint: if an object implements an interface, you cast it into that interface type:

```
MyObject o = new MyObject();
Comparable<MyObject> c = (Comparable<MyObject>)o;
```

This gives a warning, but don't worry about that.

Thus, if I have `MyObject o1`, then this allows me to execute:

```
c.compareTo(o1);
```

If they are not `Comparable`, then there should be a `Comparator` field in the `BinarySearchTree` class that can be used to call `compare()` on the values of any two `TreeNode` objects.