Prepared by Linan Qiu <lq2137@columbia.edu>

Maps

Pair

Here's how you think about maps: **maps are dictionaries**. The dictionaries we know (the books) essentially consist of pairs. Each of those pairs (in the abstract) consist of a word and its definition. We can model it in Java like this:

```
// Pair.java

public class Pair {
  public String word;
  public String definition;

  public Pair(String word, String definition) {
    this.word = word;
    this.definition = definition;
  }
}
```

This seems rather straightforward.

Now when we search for a word in the dictionary, do we search using the `word` or the `definition`? We almost definitely use the `word`. Hence, the `word` is like a **key** that we use to find the pair. The `definition` is then a **value** that we attach to each key.

We can thus rewrite our `Pair` in generics as such:

```
// Pair.java

public class Pair<K, V> {
  public K key;
  public V value;

  public Pair(K key, V value) {
    this.key = key;
    this.value = value;
  }
}
```

What we have is essentially a `Pair` that can hold on to two variables, each of (potentially) different types. This is basically a more general dictionary – `key`s do not have to be strings, nor do `value`s.

Now in a dictionary, pairs are sorted by the alphabetical order of the `word`. The `definition` does not factor into the order at all. We want to model this into our `Pair` class as well. Essentially, we have to make the entire `Pair` class implement `Comparable`, but the `compareTo` method only compares the `key` of a pair to that of another pair.

```java
// Pair.java

public class Pair<K extends Comparable<? super K>, V> implements Comparable<Pair<K, V>> {
  public K key;
  public V value;

  public Pair(K key, V value) {
    this.key = key;
    this.value = value;
  }

  @Override
  public int compareTo(Pair<K, V> o) {
    return key.compareTo(o.key);
  }
}
```

And now, we have a `Pair` that, say when sorted using `Collections.sort`, will produce an ordering based **only** on the `key` variable.

Implementation of Maps

Now that we have our little `Pair` capsule that represents each entry in our dictionary, we have to think about how to implement our dictionary.

Essentially, with a dictionary, we can perform **two** functions;

- `put(K key, V value)` adds a pair of `key` and `value` to the dictionary. Notice that we are not passing `put` a `Pair`. This is because `Pair` is essentially an internal class – something like the nodes of a binary tree or a node of a linked list. If a `Pair` with the given `key` already exists within the dictionary, we **overwrite** the original value with the new one. This is akin to adding a word to the dictionary with a definition or overwriting the definition of a word.
- `get(K key)` gets the `value` of a pair with the `key` given. This is akin to looking up the definition of a word in the dictionary. This should return `null` if the `key` does not exist within the dictionary.

Using this, we can create a `Map` interface that we demand our implementations to well... implement.

```java
// Map.java

public interface Map<K, V> {
  public void put(K key, V value);

  public V get(K key);
}
```