# Java `Map` and `Set` collections

- Exceptions poll from last time.
- Comparator example (unofficial HW problem from last time)
- Java `Set` container
  - idea
  - interface
- Java `Map` container
  - idea
  - interface
  - concordance example
  - (Next lecture: iterating over a map)

# Announcements

- Next set of course material videos available on binary search and search trees. Due by Tu 11/5 lecture.
- Reminder: MT 2 on Tues, 11/12 (2 weeks away)
- MT 2 Sample exams available

# Additional example of implementing an interface

- Problem: sort an array of **`Rectangle`**'s in increasing order by area.
- Do not implement your own sort method!

```
public static void sortIncrByArea(
                              Rectangle[] rects) {
    Arrays.sort(
```

# Java Collections

- **`Collection`** is an interface in Java
- Linear collections:

  **`ArrayList`**, **`LinkedList`**, **`Stack`**, **`Queue`**

  – ordering of elements depended on order and type of insertion (i.e. by the client)
- Two others today: **`Set`** and **`Map`**
  – ordering is determined internally by the class based on *value* of the element
  – goal: want Set or Map to be able to efficiently search by that value.

# Set ADT

(ADT = abstract data type)

Operations:

- add an element (no duplicate elements added)
- remove an element
- ask if an object is in the set
- list all the elements
  - (order of visiting depends on the kind of set created)

# Simple applications of Sets

- Determine the number of different words in a text file.

- Spell-checker (Ex from Section 15.3.2 of text)

# Java **`Set`** interface

- Two implementations:

```
Set<ElmtType> s = new HashSet<ElmtType>();
```

  - fastest. for when you don't care about order when iterating, or if you don’t need to iterate.
  - **`ElmtType`** must support **`equals()`** and **`hashCode()`**

```
Set<ElmtType> s = new TreeSet<ElmtType>();
```

  - for when you need to visit element in sorted order.
  - **`ElmtType`** must implement **`Comparable`** (has **`compareTo`**)

- Normally use *interface* type for object variable. E.g.,

```
Set<String> uniqueWords =
                          new TreeSet<>();
```

# Java **`Set`** interface (cont.)

```
Set<String> mySet =
        new TreeSet<String>();
```
creates empty set

```
mySet.add("the");
```
if **wasn't** there, adds it and returns true,
o.w., returns false and set unchanged

```
mySet.remove("blob");
```
if it **was** there, removes it and returns true,
o.w., returns false and set unchanged

```
mySet.contains("the")
```
returns true iff "the" is in the set

```
size()   isEmpty()
```

# Iterating over a Set

- **`Iterator`** is also an interface
- Order elements visited depends on kind of Set involved.
- Can iterate over other Collections like we did with LinkedList. E.g.,

```
Set<String> mySet = …;
...
Iterator<String> iter =
                    mySet.iterator();
while (iter.hasNext()) {
    String word = iter.next();
    System.out.println(word);
          // or do something else with it
}
```

# Who owns elements in a Set?

- Like with **`ArrayList`**/**`LinkedList`** elements are not "owned" by the set: i.e., no defensive copy made. (fine for those classes)
- safest if ElmtType is an immutable type (e.g., **`String`**, **`Integer`**)

if not . . .

- Unsafe to mutate element contents while it's in the Set: recall, organized by the value of elements
- example next slide . . .

# Illustration of invalidating a Set by mutating a value while it's part of the Set

```
Set<Point> setOfPoints = . . .
Point p = new Point(3, 5);
setOfPoints.add(p);
. . .
p.translate(10, 20); // BAD -- invalidates set
```

# Another example of invalidating the Set

- While iterating over the set:
- Note: iterator **`next()`** returns a reference to the element:

```
Set<Point> setOfPoints = . . .
. . .
int x = 4;
int y = 1;
Iterator<Point> iter =
                    setOfPoints.iterator();
while (iter.hasNext()) {
    Point p = iter.next();
    p.translate(x, y); // BAD -- invalidates set
    x++;
    y++;
}
```

# How many different words in a file?

```
public static int numUnique(Scanner in) {
```

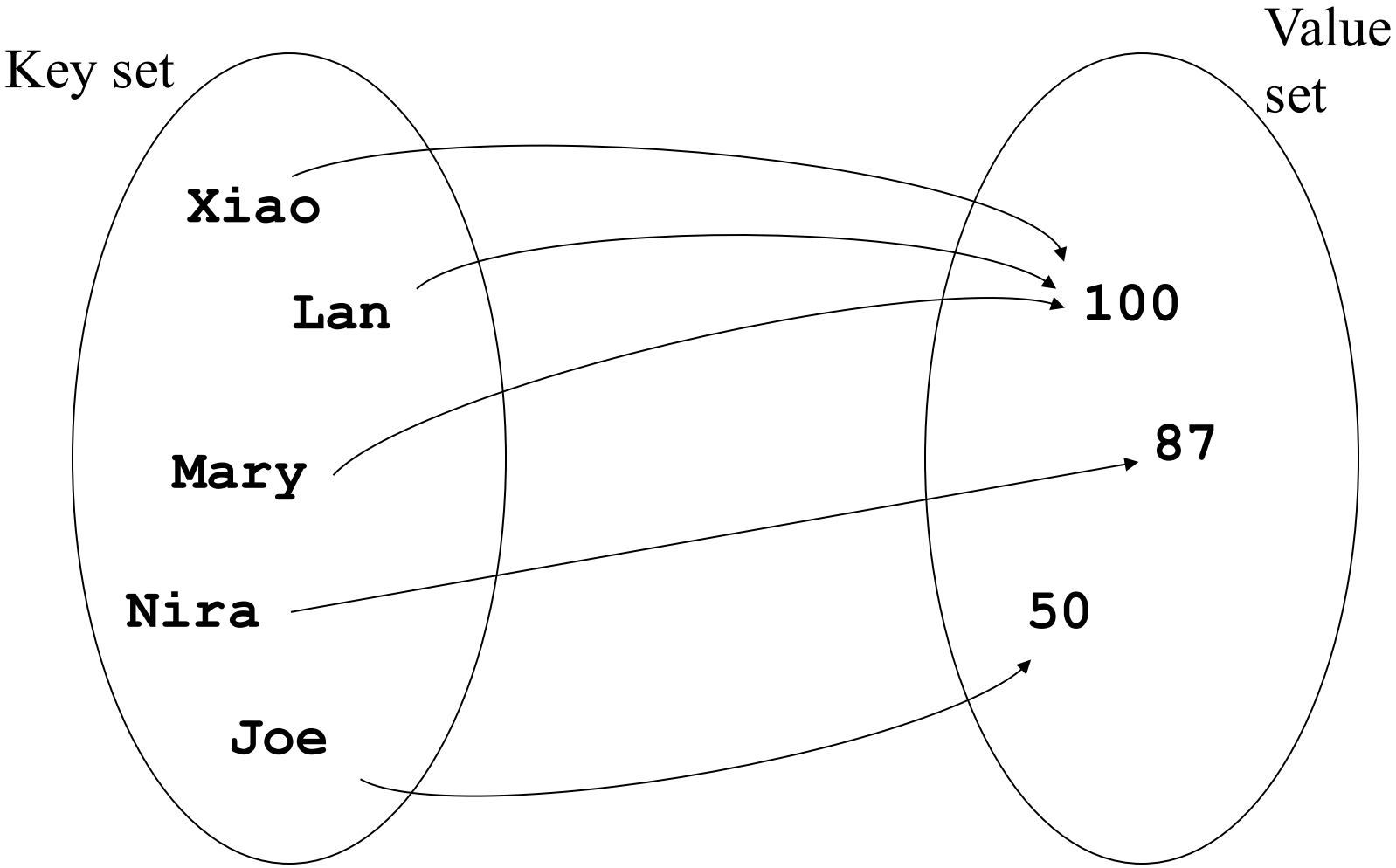
# Map ADT

- A map stores a collection of (key,value) pairs
- keys are unique: a pair can be identified by its key

Operations:

- add a new (key, value) pair (called an *entry*)
- remove an entry, given its key
- lookup a value, given its key
- update the value part of an entry, given its key
- list all the entries
  - (order of visiting depends on the kind of map created)

# Example: map of students and their scores



# Java **`Map`** interface

- Creation is same as Set, but *two* type parameters for generic class.

```
Map<KeyType, ValueType> map =
      new HashMap<KeyType,ValueType>();
```

  - fastest. for when you don't care about order when iterating, or if you don't need to iterate.
  - **`KeyType`** must support **`equals()`** and **`hashCode()`**

```
Map<KeyType, ValueType> map =
      new TreeMap<KeyType,ValueType>();
```

  - for when you need to visit element in sorted order by keys.
  - **`KeyType`** must implement **`Comparable`** (has **`compareTo`**)

# Java **`Map`** interface (cont.)

- Create an empty map:

```
Map<String, Integer> scores =
            new TreeMap<String, Integer>();
```

- Note: **`put`** operation can be used in two ways:
- Suppose we do the two operations below in sequence:

```
scores.put("Joe", 98); // inserts
```

if key wasn't there, adds it and returns **`null`**,
o.w., returns the old value that went with this key

```
scores.put("Joe", 100); // updates
```

changes Joe's score to 100. if "Joe" hadn't been there before, this would have added him.

# Java **`Map`** interface (cont.)

```
Map<String, Integer> scores =
        new TreeMap<String, Integer>();
scores.remove("Joe");
```

if key was there, removes it and returns the value that went with this key, o.w., returns **`null`** and map is unchanged

```
Integer score = scores.get("Joe");
```

return the value that goes with "Joe", or **`null`** if "Joe" is not in the map

```
boolean isThere = scores.containsKey("Joe");
```

# More about **`get`**

- Can't just use return value of **`get`** as valid object reference, because it returns **`null`** sometimes:

```
Map<String, Integer> scores = new HashMap<>();
int score = scores.get("Joe");  // crashes
```

*instead…*

```
Integer scoreI = scores.get("Joe");
if (scoreI != null) {
   int score = scoreI;  // safe to unwrap Integer
}
```

# **`Map`** seen as an array

- Map ADT is sometimes called an *associative array*

```
System.out.println(scores.get("Joe"));
```

- ArrayList index syntax, but it's not random access
- But it is fast:
  - TreeMap: get, put, remove O(log n) each.
  - HashMap: get, put, remove O(1) each (!)
- E.g., Need an "array" indexed by a String?

. . . use a Map

# Example: concordance

Problem: find the number of occurrences of each word in a text document.

- Why?
- (Variation also finds the page numbers or line numbers where those words occur in the document.)

# Example: concordance (cont.)

- Similar to finding frequencies of student scores (from earlier in the semester):

```
// sample scores: 72 99 84 99 72 85 72 80
// scores are all in range [0..100]

int[] freq = new int[101];

for each score
  freq[score]++;
```

- Can we use an array in the same way for this problem?:

  Find the number of occurrences of each word in a text document.