CSE 373

Google Guava Collection Library

slides created by Marty Stepp
http://www.cs.washington.edu/373/

US Presidents data

- Suppose we have data about US Presidents of the form:

```
// president    party  years     vp
George Washington:I:1789:1797:John Adams
John Adams:F:1797:1801:Thomas Jefferson
```

- How can we answer questions such as the following:
  - Who was president in 1873?
  - Which party has had the most Presidents?
  - Who are all the Presidents with the first name "William"?
  - Who was James Garfield's VP?
    Who was president when Adlai Stevenson was VP?
  - How many total years has a Republican been President?

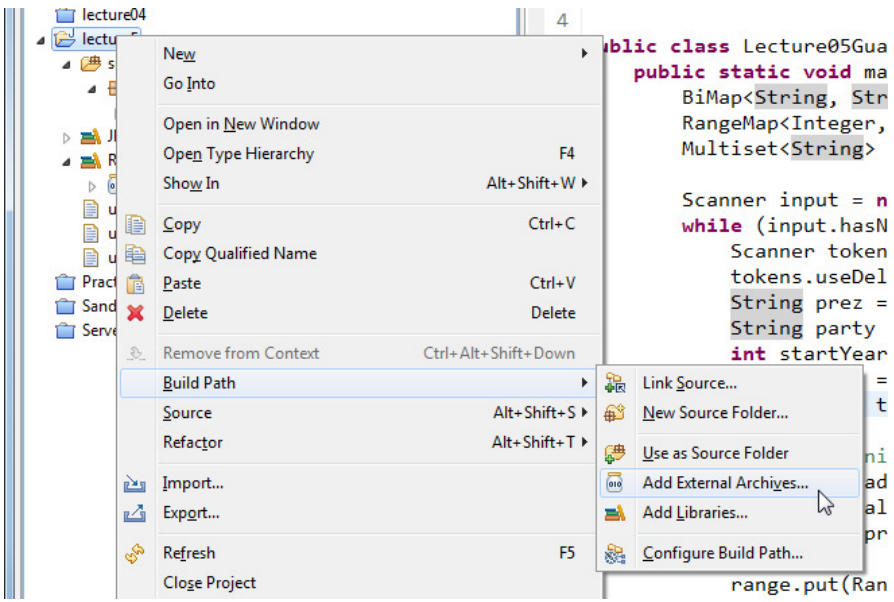
- Java's collection classes can answer these questions, but not without some effort. Guava can make it easier.

Google's Guava library

- **guava**: Google's add-on library for Java.
  - used and developed internally for years by Google developers
  - now released as free open source software
  - adds many useful classes and features to Java libraries
    - avoiding null, exception handling, new **collections**, functional programming, simple I/O, event processing, math functions, reflection, ...
  - download from http://code.google.com/p/guava-libraries/
  - API docs at http://docs.guava-libraries.googlecode.com/git-history/release/javadoc/index.html

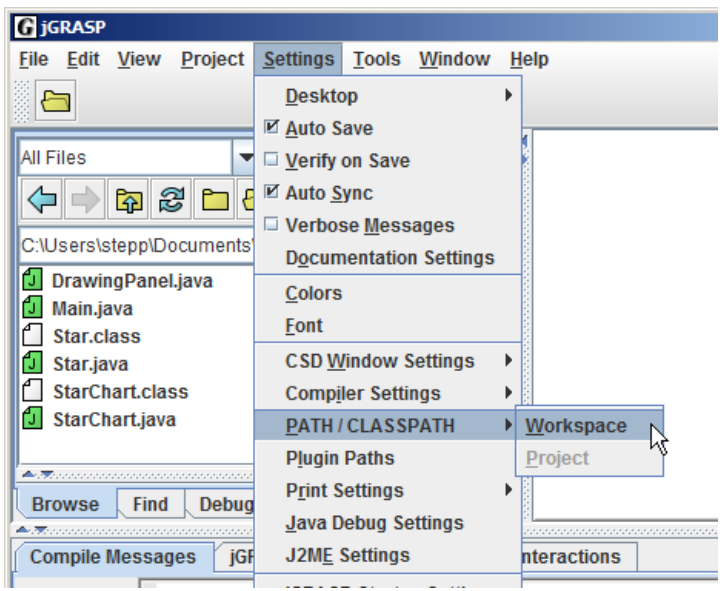
Using Guava library

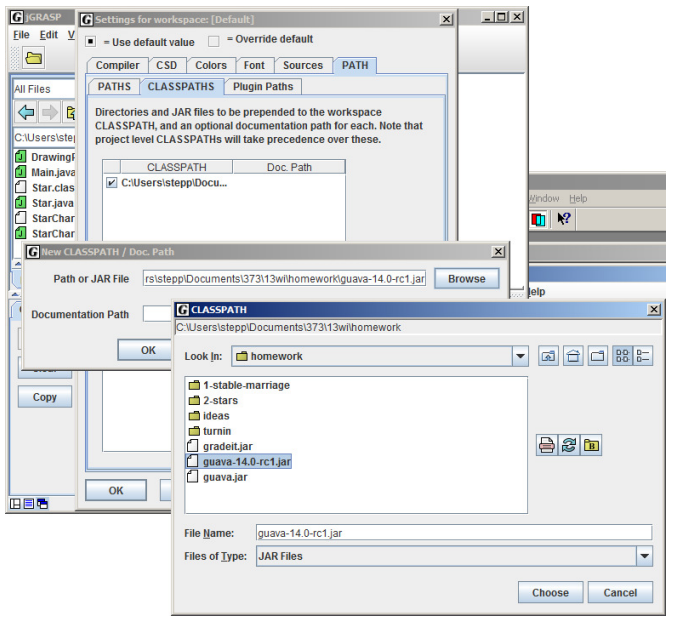
- Download Guava JAR from: http://code.google.com/p/guava-libraries/
- Attach Guava to your Eclipse project:
  - right-click project, choose **Build Path** → **Add External Archives ...**
  - browse to the Guava JAR and select it
  - in your code: `import com.google.common.collect.*;`



Guava and jGRASP

- Click **Settings** → **PATH/CLASSPATH** → **Workspace**.
- A dialog box pops up. Click the **CLASSPATHS** tab.
- Another box pops up. Click **New**. Click **Browse**.
- Now a file browser pops up.
  Go find the location of your Guava .jar file and select it.
- Click **OK** until you're back at your code.

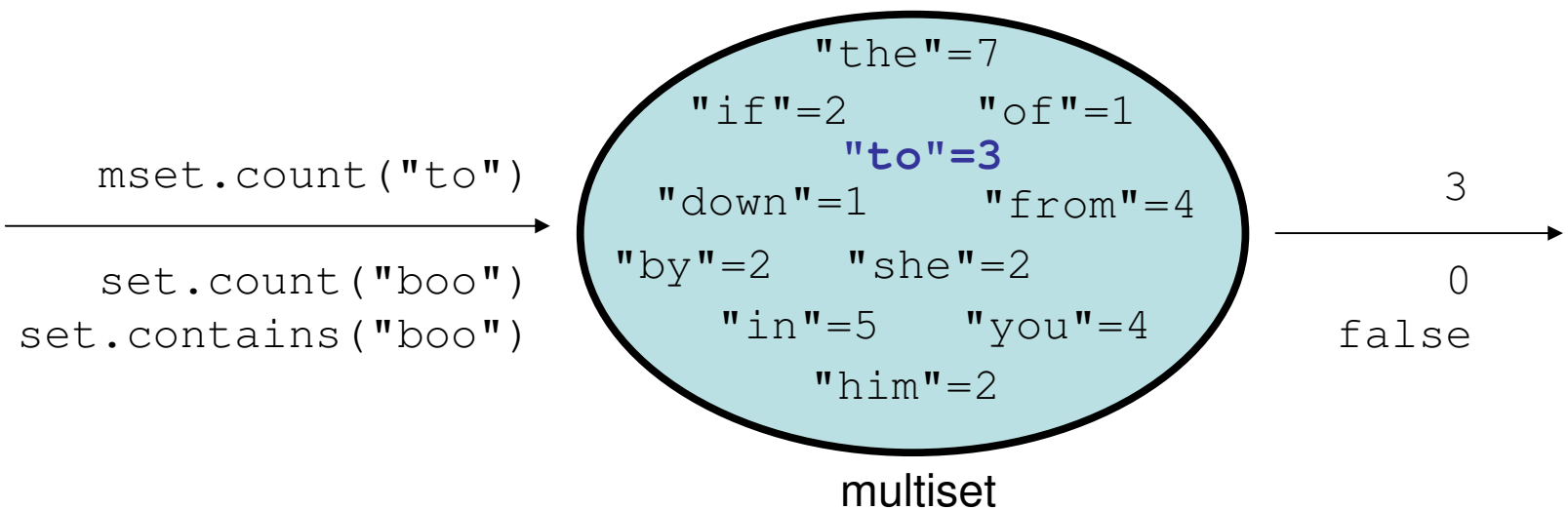




Multiset

- **multi-set**: a set of counters; also called a "bag"
  - counts the # of times each unique value was added
  - meant to replace `Map<K, Integer>`
  - implementations: `HashMultiset`,`LinkedHashMultiset`,`TreeMultiset`

```
// convention: construct using create() method
Multiset<String> mset = HashMultiset.create();
```



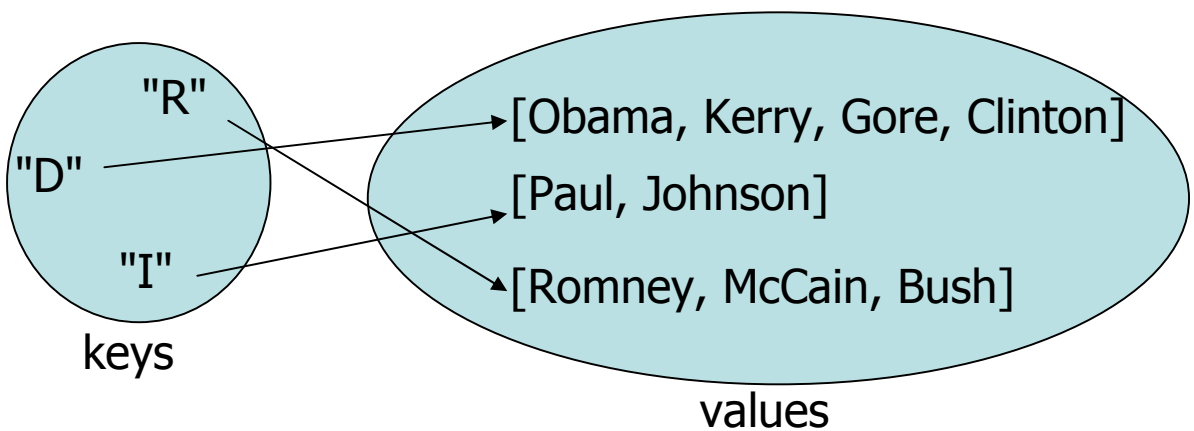
Multiset methods

class.`create()` **class**.`create(`**collection**`)`	creates a new empty multiset, or one based on the elements of a collection
`add(`**value**`)` `add(`**value**`, `**count**`)`	adds 1 occurrence of value to collection; or adds the given # of occurrences
`contains(`**value**`)`	true if set contains ≥ 1 occurrence of value
`count(`**value**`)`	returns # of occurrences of value; 0 if not found
`iterator()`	an object to examine all values in the set
`remove(`**value**`)` `remove(`**value**`, `**count**`)`	removes 1 occurrence of the given value; or removes the given # of occurrences
`setCount(`**value**`, `**count**`)`	causes the given value to have the given count
`size()`	returns sum of all counts
`toString()`	string such as `"[a x 4, b x 2, c]"`
`elementSet()`, `entrySet()`	collection views of the multiset

Multimap

- **multi-map**: a map from keys to collections of values
  - meant to replace `Map<K, Set<V>>` or `Map<K, List<V>>`
  - implementations: `ArrayListMultimap`, `LinkedListMultimap`, `HashMultimap`, `LinkedHashMultimap`, `TreeMultimap`

```
// political party -> people in it
Multimap<String, String> mmap = TreeMultimap.create();
mmap.put("D", "Gore");
mmap.put("D", "Clinton");
```



Multimap methods

class.`create()` **class**.`create(`**map**`)`	creates a new empty multimap, or one based on the elements of a map
`clear()`	removes all key/value pairs
`containsKey(`**key**`)`	returns true if the given key is stored
`get(`**key**`)`	returns collection of values associated with key
`put(`**key**`, `**value**`)`	adds value to this key's collection
`putAll(`**key**`, `**collection**`)`	adds all given values to this key's collection
`remove(`**key**`, `**value**`)`	removes value from this key's collection
`removeAll(`**key**`)`	removes all values associated with this key
`size()`	returns number of key/value pairs
`toString()`	string such as `"{a=[b, c], d=[e]}"`
`asMap(), keys(),` `keySet(), values()`	various collection views of the map's data

Choosing a Multimap

- The `Multimap` has two sub-ADT interfaces:
  - **`ListMultimap`** → `ArrayListMultimap, LinkedListMultimap`
  - **`SetMultimap`** → `Hash, LinkedHash, TreeMultimap`

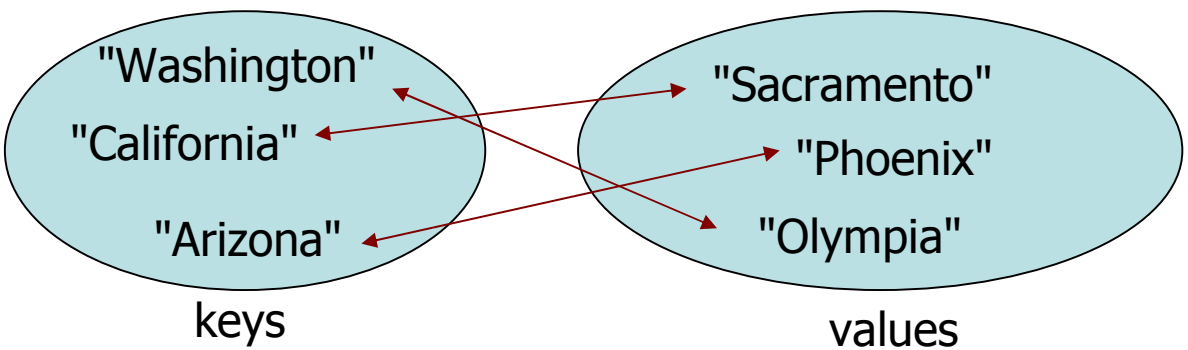
  - If you need list-specific methods, declare it as a `ListMultimap`.

```
ListMultimap<String, String> mmap =
        ArrayListMultimap.create();
mmap.put("D", "Gore");
mmap.put("D", "Clinton");
System.out.println(mmap.get("D").get(0);  // Gore
System.out.println(mmap.get("D").get(1);  // Clinton
```

BiMap

- **bi-map**: a two-directional map
  - for data where a $\rightarrow$ b and also b $\rightarrow$ a in symmetry
  - avoids need to try to "invert" a map or store an inverse map
  - implementations: `HashBiMap`

```
// state <--> state capital
BiMap<String, String> bmap = HashBiMap.create();
mmap.put("Arizona", "Phoenix");
mmap.put("Washington", "Olympia");
```



BiMap methods

- all methods from `Map` are present as well
  - `clear`, `containsKey`, `containsValue`, `equals`, `get`, `isEmpty`, `keySet`, `put`, `putAll`, `remove`, `size`, `toString`

class `.create()` **class** `.create(`**map**`)`	creates a new empty bi-map, or one based on the elements of a map
`inverse()`	returns `BiMap<V, K>` in opposite direction
`values()`	returns *set* of all values

Table

- **table**: a two-dimensional (key+key) / value structure
  - meant to replace `Map<R, Map<C, V>>`
  - a *map* stores pairs of form (K, V) where only K is known later;
    a *table* stores triples of form (R, C, V) where R,C are known later
  - implementations: `HashBasedTable`, `TreeBasedTable`, `ArrayTable`

```
// (name + SSN => age)
Table<String, String, Integer> table =
TreeBasedTable.create();
table.put("Marty Stepp", "597-24-6138", 29);
```

name	SSN	age
Marty Stepp	597-24-6138	29
Stuart Reges	703-34-1593	84

Table methods

class`.create()`	creates a new empty table, etc.
`cellSet()`	set of all (R, C, V) triples
`clear()`	remove all values
`column(`**C**`)`	returns column for given key as Map<R,V>
`contains(`**R, C**`)`	true if table has a mapping for the given keys
`containsRow(`**R**`),` `containsColumn(`**C**`)`	true if table has any mapping that includes the given row or column key
`get(`**R, C**`)`	returns value for the given keys, or null
`isEmpty()`	true if there are no values
`put(`**R, C, V**`)`	stores (R, C, V) triple in the table
`putAll(`**table**`)`	adds all of the given table's data to this one
`remove(`**R, C**`)`	removes any value mapped from the given keys
`row(`**R**`)`	returns row for given key as a Map<C,V>
`size()`	number of triples in table
`toString()`	string such as `"{a={b=c, d=e},f={g=h}}"`

RangeSet

- **range set**: a group of comparable ranges of values
  - like a set, but you can add an entire range at a time
  - implementations: `TreeRangeSet`

```
// teenagers and old people
RangeSet<Integer> ages = TreeRangeSet.create();
ages.add(Range.closed(13, 19));
ages.add(Range.atLeast(65));
System.out.println(rset.contains(15));    // true
System.out.println(rset.contains(72));    // true
```

< 0	0 ... 12	13 ... 19	20 ... 64	≥ 65 ...

RangeSet methods

class`.create()`	creates a new empty range set, etc.
`add(`**range**`)`	adds the given range of values
`addAll(`**rangeset**`)`	adds all ranges from the given set
`clear()`	removes all ranges
`encloses(`**range**`)`	true if set contains the entire given range
`enclosesAll(`**rangeset**`)`	true if set contains all ranges in given set
`isEmpty()`	true if there are no ranges
`remove(`**range**`)`	removes the given range of values
`span()`	a Range representing all values in this set
`subRangeSet(`**range**`)`	subset containing relevant ranges
`toString()`	string such as `"[1..3], (6..65]"`

Specifying ranges

- Specify a range of values by calling static methods of the `Range` class, each of which returns a `Range` object.

`Range.closed(`**min, max**`)`	[min .. max] including both endpoints
`Range.open(`**min, max**`)`	(min .. max) excluding min and max
`Range.closedOpen(`**min, max**`)`	[min .. max) include min, exclude max
`Range.openClosed(`**min, max**`)`	(min .. max] exclude min, include max
`Range.atLeast(`**min**`)`	[min .. ∞) including min
`Range.greaterThan(`**min**`)`	(min .. ∞) excluding min
`Range.atMost(`**max**`)`	(-∞ .. max] including max
`Range.lessThan(`**max**`)`	(-∞ .. max) excluding max
`Range.all()`	all possible values, (-∞ .. ∞)
`Range.singleton(`**value**`)`	[value]; just a single value

RangeMap

- **range map**: like a range set, but stores (*range*, *value*) pairs
  - implementations: `TreeRangeMap`

```
// body mass index -> description
RangeMap<Double, String> bmi =
        TreeRangeMap.create();
bmi.put(Range.lessThan(18.5), "underweight");
bmi.put(Range.closedOpen(18.5, 25.0), "normal");
bmi.put(Range.closedOpen(25.0, 30.0), "overweight");
bmi.put(Range.atLeast(30.0), "obese");
System.out.println(bmi.get(27.1));   // "overweight"
```

< 18.5	18.5 .. 25.0	25.0 .. 30.0	≥ 30.0
underweight	normal	overweight	obese

RangeMap methods

class`.create()`	creates a new empty range map, etc.
`put(`**range**`, `**value**`)`	adds range/value pair
`putAll(`**rangemap**`)`	adds all range/value pairs from given map
`clear()`	removes all ranges
`get(`**key**`)`	returns value for range containing key
`isEmpty()`	true if there are no ranges
`remove(`**range**`)`	removes all values in the given range
`span()`	a Range representing all keys in this set
`subRangeMap(`**range**`)`	submap containing relevant ranges
`toString()`	string such as `"{[1..3]=a, (6..65]=b}"`

Other cool features

- `Collections2`: utility methods related to all collections
- `Lists`: utility methods related to lists
- `Sets`: utility methods related to sets
- `Queues`: utility methods related to queues
- `Multisets`, `Multimaps`: utility methods related to multiset/map
- `Tables`: utility methods related to tables
- `Iterables`: utility methods related to collections and for-each
- `Iterators`: utility methods related to iterators and iteration
- `Ordering`: easy-to-create comparable and comparator orders
- `Immutable*`: collections that cannot be modified

- see also: http://code.google.com/p/guava-libraries/wiki/CollectionUtilitiesExplained

Collection views

- **collection view**: a collection that is based on another collection
  - links back to the original collection; changes to one affect the other
  - *important*: does not make deep copy of collection (O(1), not O(*N*))
  - `Arrays`
    - `public static List<T> asList(T[] elements)`
      Returns a `List` view backed by an array or group of parameter values.
  - `List<E>`
    - `public List<E> subList(int start, int end)`
      Returns a view of indexes [start, end) of this list.
  - `Map<K, V>`
    - `public Set<K> keySet()`
      Returns a `Set` view of all keys in the given map.
    - `public Collection<V> values()`
      Returns a `Collection` view of all values in the given map.

Sorted Views

- `SortedSet<E>` (a `TreeSet` is a `SortedSet`)
  - `public SortedSet<E> subSet(E start, E end)`
    `public SortedSet<E> headSet(E end)`
    `public SortedMap<E> tailMap(K start)`
    Returns a set view of range [start .. end), (-∞ .. end), or [start .. ∞).
- `SortedMap<K, V>` (a `TreeMap` is a `SortedMap`)
  - `public SortedMap<K, V> subMap(K start, K end)`
    `public SortedMap<K, V> headMap(K end)`
    `public SortedMap<K, V> tailMap(K start)`
    Returns a map view of range [start .. end), (-∞ .. end), or [start .. ∞).
- `RangeSet<E>`
  - `public RangeSet<E> subRangeSet(Range<E> view)`
    Returns a set view of the given range of element values.
- `RangeMap<K, V>`
  - `public RangeMap<K, V> subRangeMap(Range<E> view)`
    Returns a map view of the given range of key/value pairs.

Immutable views

- `Collections`
  - `public static List<E> unmodifiableList(List<E> list)`
    `public static Map<K, V> unmodifiableMap(Map<K, V> map)`
    `public static Set<E> unmodifiableSet(Set<E> set)`
    `public static Collection<E> unmodifiableCollection(Collection<E> set)`

    Returns an appropriate view of the given collection that cannot be modified (calls to methods such as `set`, `remove`, `clear`, etc. will throw an `UnsupportedOperationException`).

- Guava
  - many provided immutable collections, e.g. `ImmutableBiMap`, `ImmutableMultimap`, `ImmutableMultiset`, `ImmutableRangeSet`, `ImmutableRangeMap`, `ImmutableTable`, …
  - useful when you want to share a collection without it being changed

Collection view examples

```
// sort a sub-portion of a list
List<String> list = new ArrayList<String>();
list.add("z"); list.add("y"); list.add("x");    // 0 1 2
list.add("c"); list.add("b"); list.add("a");    // 3 4 5
Collections.sort(list.subList(1, 4));    // [z,c,x,y,b,a]
someUnsafeMethod(Collections.unmodifiableList(list));

// delete three students' grades
Map<String, Double> gpa = ...;          // fill with data
gpa.keySet().removeAll(Arrays.asList("Ned", "Rod", "Todd"));

// body mass index -> description  (from prior slide)
// delete certain body mass ranges from the map
RangeMap<Double, String> bmi = TreeRangeMap.create();
bmi.subRangeMap(Range.closed(10, 20)).clear();
```