CSE 373 Section Handout #3
Syntax Reference

Common Methods

(methods found in every Guava collection)

class`.create()`
class`.create(`**collection**`)`
	constructs a new empty collection,
	or one based on the contents of another collection

`clear()`	remove all elements from the collection
`isEmpty()`	returns `true` if the collection contains no elements
`size()`	returns the number of elements in the collection
`toString()`	returns a string representation of the collection and its elements

`Multiset<E>` Methods

(a collection of counters of occurrences of all values that have been added)

- implementations: `HashMultiset`, `LinkedHashMultiset`, `TreeMultiset`

`addAll(`**collection**`)`	adds all elements from the given collection to this set

`add(`**value**`)`
`add(`**value, count**`)`
	adds 1 occurrence of the given value, or many occurrences

`contains(`**value**`)`	returns `true` if the set contains the given value at least once
`count(`**value**`)`	returns count of occurrences of the given value

`remove(`**value**`)`
`remove(`**value, count**`)`
	removes 1 occurrence of the given value, or many occurrences

`setCount(`**value, count**`)`	sets the given value to have the given number of occurrences

`elementSet(),`
`entrySet()`
	views of the elements (with duplicates included multiple times) or entry counts (as `Multiset.Entry<E>` objects)

`Multimap<K, V>` Methods

(A mapping from keys to collections of values)

- ADT sub-interfaces: `ListMultimap`, `SetMultimap`
- implementations: `HashMultimap`, `LinkedHashMultimap`, `TreeMultimap`,
  `ArrayListMultimap`, `LinkedListMultimap`

`containsEntry(`**k, v**`)`	returns `true` if the given key contains the given value in its collection
`containsKey(`**key**`)`	returns `true` if the given key maps to any values
`get(`**key**`)`	returns this key's associated collection (empty if no values added)
`put(`**key, value**`)`	adds the given value to this key's collection
`putAll(`**key, collection**`)`	adds all values from the given collection to this key's collection
`remove(`**key, value**`)`	removes the given value from this key's collection
`removeAll(`**key**`)`	removes all values from the given key's collection

`keys(), keySet(),`
`values()`
	collection views of the map or its keys/values

`BiMap<K, V>` Methods

(A reversible bi-directional mapping from unique keys to unique values)

- implementations: `HashBiMap`
- contains all Map methods: containsKey, containsValue, equals, get, keySet, put, putAll, remove...

`inverse()`	returns a `BiMap<V,K>` view whose keys are this `BiMap`'s values and vice versa

CSE 373 Section Handout #3
Syntax Reference, continued

`Table<R, C, V>` Methods *(A two-dimensional key+key -> value look-up; like a map of maps)*

- implementations: `HashBasedTable`, `TreeBasedTable`, `ArrayTable`

`column(`**C**`)`	returns a `Map<R,V>` view of values in the given column
`contains(`**R, C**`)`	returns true if the table contains an entry at the given row/column
`containsRow(`**R**`)`, `containsColumn(`**C**`)`	returns true if the table contains any entries in the given row or column
`get(`**R, C**`)`	returns the value associated with the given row/column pair (or null)
`put(`**R, C, V**`)`	stores the given value associated with the given row/column pair
`putAll(`**table**`)`	stores all values from the given table into this one
`remove(`**R, C**`)`	removes the value associated with the given row/column pair
`row(`**R**`)`	returns a `Map<C,V>` view of values in the given row
`columnKeySet(), columnMap(),cellSet(), rowKeySet(), rowMap()`	collection views of all cells, rows, and columns in the table

`RangeSet<E>` Methods *(A set of non-overlapping comparable ranges of values)*

- implementations: `TreeRangeSet`

`add(`**range**`)`	adds the given range of values to the set
`encloses(`**range**`)`	returns true if the given range is entirely contained in the set
`remove(`**range**`)`	removes the given range of values from the set
`span()`	returns a minimal `Range` enclosing all keys in this map
`subRangeSet(`**range**`)`	returns a `RangeSet` view of this set's data within the given range

`RangeMap<K, V>` Methods *(A mapping from comparable ranges of keys to values)*

- implementations: `TreeRangeMap`

`get(`**key**`)`	returns the value associated with the range containing the given key
`put(`**range, value**`)`	associates the given range of keys with the given value
`remove(`**range**`)`	removes the value(s) associated with the given range of keys
`span()`	returns a minimal `Range` enclosing all keys in this map
`subRangeMap(`**range**`)`	returns a `RangeMap` view of this map's data within the given range

`Range<E>` Methods *(Objects that represent ranges of values)*

`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

CSE 373 Section Handout #3

Choosing a Collection

1. **Guava Collections**
   What is a good Guava collection to use to represent each of the following? Justify your answer.
   a) an address book where you can look up someone's social security number by their name *or vice versa*
   b) a car owners' book where you can look up someone's car license plate number*(s)* by their name
   c) a frequent-buy system for a coffee shop, where a customer gets a free coffee after every 10th purchase
   d) an Urban Dictionary of phrases, where each phrase can have one or more definitions
   e) a collection of vote counts by district, where given a candidate and district #, we can find his votes there
   f) a collection of word lengths, where it is easy/efficient to answer the question, "What are all words in the English dictionary of length N?"
   g) TV ratings data, organized by age demographics (children under 18, 18-25, 25-40, 40-65, 65+)
   h) data about students' grades, where given a course name and a student's name, you can quickly discover that student's grade (0.0 to 4.0) in that course
   i) an index in a textbook, where you want to know what page number(s) each specific term appears on

Multiset / Multimap

2. **`tallyLetters`**
   Write a method named `tallyLetters` that accepts a string as a parameter and that returns a `Multiset` of counts of occurrences of each alphabetic letter (`Character`) in the string, case-insensitively. For example, for the string `"Hello How is he DOING?"`, return the multiset `[D, E, G, H x 3, I x 2, L x 2, N, O x 3, W]`. Ignore any characters that are not letters (A-Z).

3. **`removeOddCount`**
   Write a method named `removeOddCount` that accepts a `List` of strings as a parameter and removes any element value that occurs an odd number of times from the list. For example, in the list `[a, b, a, c, a, d, d, b, b, b, e, e, b]`, the strings `a`, `b`, and `c` occur an odd number of times (3, 5, 1 times respectively), so your method should change the list to `[d, d, e, e]`. Use one Guava `Multiset` as auxiliary storage.

4. **`sockPairs`**
   A common complaint is that people can't find one of the two socks in a matching pair after they do the laundry. So in this problem you'll write a method named `sockPairs` that looks at a collection of socks in the laundry and determines whether all types of socks occur exactly twice. Your method accepts an array of strings as a parameter; each string represents a type of sock. Your method should examine the list and return `true` if every unique string in the list occurs exactly 2 times, or `false` if any string does not occur exactly 2 times. An empty array should cause your method to return `true`. For example, if passed the array `[green, white, green, polkadot, red, black, red, white, black, polkadot]`, your method should return `true` because every string occurs exactly twice. If passed `[green, white, green, green, white, green, red]`, your method should return `false` because `green` occurs 4 times and `red` occurs only once.

5. **`byStartingLetter`**
   Write a method named `byStartingLetter` that accepts an array of words (strings) as a parameter and that returns a `Multimap` of the words, keyed by their starting letter (`Character`), case-insensitively. For example, if the array `["Hello", "how", "is", "dear", "old", "Dad", "DOING?"]` is passed, return the multimap `{D=[dad, dear, doing], H=[Hello, how], I=[is], O=[old]}`.

CSE 373 Section Handout #3

BiMap

6. **`rapperNames`**

A company is hiring new employees, but they do not want to hire anyone who is a known rapper, because rappers are questionable characters. Write a method named `rapperNames` that accepts two parameters: a `BiMap` from people's names to their rapper nicknames, and a `Set` of possible interview candidates (strings). We don't want to interview the ones who are secretly rappers. You should return a new `Set` of which candidates the company would like to interview. If an interview candidate string is the real name or the rapper nickname of a known rapper, he/she should not be interviewed. For example, given this data:

```
rappers:    {Marshall Mathers=Eminem, Shad Moss=Bowwow, Sean Combs=Diddy,
             Trevor Smith=Busta Rhymes, Michael Diamond=Mike D}
candidates: [Jim Jones, Eminem, Sean Combs, Suzy Smith, Michael Diamond, Busta Rhymes]
```

You would return the set `[Jim Jones, Suzy Smith]` to be interviewed.

Table

7. **`sharedBirthday`**

Write a method named `sharedBirthday` that accepts an array of integers for a group of people's birthdays and returns `true` if any two people have the same birthday. Each pair of array elements represents someone's birth month and day. For example, if your method is passed the array `[3, 14, 12, 25, `**`2, 9`**`, 8, 17, 3, 28, `**`2, 9`**`, 6, 17]`, you should return `true` because the birthday of February 9 occurs twice (at index 4 and 10). Use a Guava `Table` as auxiliary storage.

8. **`xmasMoney`**

Write a method named `xmasMoney` that computes the difference between the total cost of gifts a person received, and the total money the person spent on their Xmas gifts to others. The method accepts two parameters: the person's name of interest, and a `Table` of Xmas gift prices, where the two string keys (*person1*, *person2*) map to a real number (`double`) representing the price of the gift that *person1* gave to *person2* for Xmas. It is possible that not every person gave a gift to every other person. If the person did not give or receive any gifts, return `0.0`. For example, if the name is `Mary` and the table is the following:

```
{John={Mary=50.50, Zora=10.25},
 Mary={John=24.50, Zora=10.50},
 Stan={Mary=45.00},
 Zora={John=85.75, Stan=40.00, Zora=99.25}}
```

Then you would return `60.5` because Mary received $95.50 worth of presents from John and Stan but spent only $35.00 total on her presents to John and Zora.

RangeSet / RangeMap

9. **`graders`**

Write a method named `graders` that helps the TAs figure out how many students' programs each one needs to grade. The method accepts two parameters: a `Set` of students' names to grade (strings); and a `RangeMap` where each key is a range of the alphabet (strings) and each value is the TA's name (string) who grades that range of the alphabet. Your method should return a `Multiset` of counts of how many students must be graded by each TA. For example, if the set of students and range map are the following:

```
set: [Joe, Stan, Ed, Paul, Dan, Bill, Tina, Lou, Qbert, Rob, Hank, Zaza, Karl, Mike]
rangemap: {[Aa..Ez]=Conor, [Fa..Jz]=April, [Ka..Nz]=Staci, [Oa-Sz]=Dana, [Ta-Zz]=Jake}
```

Then you would return the multiset `[April x 2, Conor x 3, Staci x 3, Dana x 4, Jake x 2]`. You may assume that every name maps to an alphabetic range that is covered by one of the TAs.

CSE 373 Section Handout #3

Classes and Objects

10. **`Date-compareTo`**

Suppose a class `Date` has been defined. Each `Date` object stores a calendar date with month, day and year components. The class includes the following members:

`private int year, month, day`	state of the date
`Date(`**year, month, day**`)`	constructs a date with given year, month, day
`getYear()`	returns the year component
`getMonth()`	returns the month component
`getDay()`	returns the day component
`toString()`	returns the date

Make `Date` objects comparable to each other using the `Comparable` interface. Write the class header and any other changes you would need to make to the code to achieve this. Dates that occur chronologically earlier should be considered "less" than dates that occur later. You may assume that dates are constructed with appropriate values, such as months between 1 and 12.

11. **`Date-equals`**

Write an `equals` method for the above `Date` class that compares dates for equality. Two dates are equal if they represent the same year, month, and day. Use the proper method signature in your method so that it overrides the default `equals` method from class `Object`. If a non-`Date` object is passed, return `false`.

12. **`Date-toString`**

Write a `toString` method for the above `Date` class that returns a string representation of dates in *yyyy/mm/dd* format, such as `"2005/01/07"`. Use the proper method signature in your method so that it overrides the default `toString` method from class `Object`. Pad month or day numbers that are less than 10 with a leading zero as shown in the example call.

13. **`MapLocation-compareTo`**

Suppose a class `MapLocation` has been defined. Each `MapLocation` object stores information about a point of interest on a map as a grid location (row, column) and name. The class has the following members:

`private char row`
`private int column`
`private String name`
	state of the map location object

`MapLocation(`**row, column, name**`)`	constructs a map location with given row/col and name
`getCoordinates()`	returns the map coordinates such as `"arboretum(B8)"`
`getName()`	returns the name of this `MapLocation`
`toString()`	returns a `String` representation of the `MapLocation`

Rows will be specified by character values in the range of A to Z inclusive and column locations will be specified as integers greater than 0, as in:

```
MapLocation loc = new MapLocation('B', 8, "arboretum");
```

Make `MapLocation` objects comparable to each other using the `Comparable` interface. Write the class header and any other changes you would need to make to the code to achieve this. `MapLocation` objects should be ordered first by increasing row and then by increasing column and then alphabetically by name.

CSE 373 Section Handout #3
Solutions

1. There is often more than one acceptable answer to questions like these. Here are our choices:
   a) `BiMap<String, Integer>`
   b) `Multimap<String, String>`
   c) `Multiset<String>`
   d) `Multimap<String, String>`
   e) `Table<String, Integer, Integer>`
   f) `Multimap<Integer, String>`
   g) `RangeMap<Integer, Double>`
   h) `Table<String, String, Double>`
   i) `Multimap<String, Integer>`

2.
```
public static Multiset<Character> tallyLetters(String s) {
    Multiset<Character> letters = TreeMultiset.create();
    for (int i = 0; i < s.length(); i++) {
        char c = s.charAt(i);
        if (Character.isLetter(c)) {
            letters.add(Character.toUpperCase(c));
        }
    }
    return letters;
}
```

3. Two solutions are shown.
```
public static void removeOddCount(List<String> list) {
    Multiset<String> counts = HashMultiset.create();
    for (String s : list) {
        counts.add(s);
    }
    for (int i = list.size() - 1; i >= 0; i--) {
        if (counts.count(list.get(i)) % 2 != 0) {
            list.remove(i);
        }
    }
}
public static void removeOddCount(List<String> list) {    // much slower version
    Multiset<String> counts = HashMultiset.create();
    for (String s : list) {
        counts.add(s);
    }
    for (String s : counts) {
        while (counts.count(s) % 2 != 0 && list.contains(s)) {
            list.remove(s);
        }
    }
}
```

4.
```
public static boolean sockPairs(String[] socks) {
    Multiset<String> sockCounts = HashMultiset.create();
    for (String sock : socks) {
        sockCounts.add(sock);
    }
    for (String sock : sockCounts) {
        if (sockCounts.count(sock) != 2) {
            return false;
        }
    }
    return true;
}
```

CSE 373 Section Handout #3
Solutions, continued

5.

```
public static Multimap<Character, String> byStartingLetter(String[] words) {
    Multimap<Character, String> result = TreeMultimap.create();
    for (String word : words) {
        result.put(word.toUpperCase().charAt(0), word);
    }
    return result;
}
```

6.

```
public static Set<String> rapperNames(BiMap<String, String> rappers,
                                      Set<String> candidates) {
    Set<String> toInterview = new HashSet<String>();
    for (String name : candidates) {
        if (!rappers.containsKey(name) && !rappers.inverse().containsKey(name)) {
            toInterview.add(name);
        }
    }
    return toInterview;
}
```

7.

```
public static boolean sharedBirthday(int[] birthdates) {
    Table<Integer, Integer, Boolean> table = HashBasedTable.create();
    for (int i = 0; i < birthdates.length - 1; i += 2) {
        int month = birthdates[i];
        int day   = birthdates[i + 1];
        if (table.contains(month, day)) {
            return true;   // duplicate birthday found
        } else {
            table.put(month, day, true);
        }
    }
    return false;
}
```

8.

```
public static double xmasMoney(String me, Table<String, String, Double> gifts) {
    double mySum = 0.0;
    for (String giver : gifts.column(me).keySet()) {   // add up gifts I received
        mySum += gifts.get(giver, me);
    }
    for (String recipient : gifts.row(me).keySet()) {  // subtract gifts I gave
        mySum -= gifts.get(me, recipient);
    }
    return mySum;
}
```

9.

```
public static Multiset<String> graders(Set<String> students,
                                       RangeMap<String, String> tas) {
    Multiset<String> result = TreeMultiset.create();
    for (String studentName : students) {
        String ta = tas.get(studentName);
        result.add(ta);
    }
    return result;
}
```

CSE 373 Section Handout #3
Solutions, continued

10.
```
public class Date implements Comparable<Date> {
    ...
    public int compareTo(Date other) {
        if (year != other.year) {
            return year - other.year;
        } else if (month != other.month) {
            return month - other.month;
        } else {
            return day - other.day;
        }

        // or, return toString().compareTo(other.toString());
    }
}
```

11.
```
public boolean equals(Object o) {
    if (o instanceof Date) {
        Date other = (Date) o;
        return year == other.year && month == other.month && day == other.day;
    } else {
        return false;
    }
}
```

12. Two solutions are shown.
```
public String toString() {
    String result = year + "/";
    if (month < 10) {
        result += "0";
    }
    result += month + "/";
    if (day < 10) {
        result += "0";
    }
    result += day;
    return result;
}

public String toString() {
    return String.format("%d/%02d/%02d", year, month, day);
}
```

13.
```
public class MapLocation implements Comparable<MapLocation> {
    ...

    public int compareTo(MapLocation other) {
        if (row != other.row) {
            return row - other.row;
        } else if (column != other.column) {
            return column - other.column;
        } else {
            return name.compareTo(other.name);
        }
    }
}
```