# Making Java Bearable with Guava


GUAVA & JAVA
EXIT
guava java
guava java

# Who is this presentation for?

- Any Java Developer not familiar with Guava
- People who have to use Java by company fiat.

# Where can I get the code?

http://www.github.com/dhinojosa/usingguava

# What is it?

- Indispensable set of utilities
- Additional and Immutable collections built upon JDK
- Open Source
- Fully Generic Collections (unlike Apache Commons)
- Continually Growing (@Beta)
- Embrace DRY principle even more!

# Joiners

```
List<String> list = new ArrayList<String>();
list.add("Manny");
list.add("Moe");
list.add("Java");
boolean started = false;
for (String item : list) {
  if (started) buffer.append(",");
  buffer.append(item);
  started = true;
}
```

```
=> Manny,Moe,Java
```

# Joiners

```
Joiner.on(',').join("Manny", "Moe", "Java")
```

```
=> Manny,Moe,Java
```

# Map Joiners

```
Map<String, String> map = new LinkedHashMap<String, String>();
map.put("New Mexico", "Santa Fe");
map.put("Texas", "Austin");
map.put("Arizona", "Phoenix");
StringBuffer stringBuffer = new StringBuffer();
boolean started = false;
for (Map.Entry<String, String> entry : map.entrySet()) {
    if (started) stringBuffer.append(", ");
    stringBuffer.append(entry.getKey());
    stringBuffer.append(" -> ");
    stringBuffer.append(entry.getValue());
    started = true;
}
stringBuffer.toString()
=> "New Mexico -> Santa Fe, Texas -> Austin, Arizona -> Phoenix"
```

# Map Joiners

```
Map<String, String> map = new HashMap<String, String>();
map.put("New Mexico", "Santa Fe");
map.put("Texas", "Austin");
map.put("Arizona", "Phoenix");

Joiner.on(", ").
    withKeyValueSeparator(" -> ").join(map)

=> "New Mexico -> Santa Fe, Texas -> Austin, Arizona -> Phoenix"
```

# Map Joiners (more concise)

```
Joiner.on(", ").
    withKeyValueSeparator(" -> ").
        join(ImmutableMap.of("New Mexico",
            "Santa Fe", "Texas", "Austin",
                "Arizona", "Phoenix"));
=> "New Mexico -> Santa Fe, Texas -> Austin,
Arizona -> Phoenix"
```

# Splitters

```
Iterable<String> items =
    Splitter.on(",").trimResults()
                    .split("Manny, Moe, Java");
List<String> itemList = Lists.newArrayList(items);

itemList.get(0) // "Manny"
itemList.get(1) // "Moe"
itemList.get(2) // "Java"
```

# Map Splitters

```
String value = "New Mexico -> Santa Fe, Texas ->
                Austin, Arizona -> Phoenix";
Map<String, String> splitKeyValues = Splitter.on(",")
                .omitEmptyStrings()
                .trimResults()
                .withKeyValueSeparator("->")
                .split(value);
```

# Preconditions

# Preconditions

```
import static com.google.common.base.Preconditions.*;


public void addGrade(Integer grade) {
    checkNotNull(grade,
       "grade cannot be null");
    checkArgument
       (grade >= 0 && grade < 101,
          "Grade must be between 0 and 101");
    this.grades.add(grade);
}
```

# Preconditions

```
Integer grade = null;
addGrade(grade);


java.lang.NullPointerException:
  Grade cannot be null
    at com.google.common.base.Preconditions
    .checkNotNull(Preconditions.java:204)
```

# Preconditions

```
import static com.google.common.base.Preconditions.*;


public void addGrade(Integer grade) {
    checkNotNull(grade,
       "grade cannot be null");
    checkArgument
       (grade >= 0 && grade < 101,
          "Grade must be between 0 and 101");
    this.grades.add(grade);
}
```

# Preconditions

```
Integer grade = 133;
addGrade(grade);


java.lang.IllegalArgumentException:
    Grade must be between 0 and 101 at
     com.google.common.base.Preconditions.
      checkArgument(Preconditions.java:88)
```

# Preconditions

```
import static com.google.common.base.Preconditions.*;

public class Book {
    private List<Integer> grades;
    ...
    public Integer getHighestGrade() {
        checkState(grades != null, "Grades
            are not set");
        checkState(grades.size() > 0, "No
            grades are entered");
        return Collections.max(this.grades);
    }
}
```

# Preconditions

```
Book book = new Book();
book.getHighestGrade();


java.lang.IllegalStateException:
  Grades are not set at
    com.google.common.base.Preconditions.
      checkState(Preconditions.java:145)
```

# Preconditions

```
import static com.google.common.base.Preconditions.*;


public class Book {
    private List<Integer> grades;
    ...
    public Integer getHighestGrade() {
        checkState(grades != null, "Grades
            are not set");
        checkState(grades.size() > 0, "No
            grades are entered");
        return
         Collections.max
            (this.grades);
    }
}
```

# Preconditions

```
Book book = new Book(Lists.
  <Integer>newArrayList());
book.getHighestGrade();
```

```
java.lang.IllegalStateException: No grades
  are entered at
    com.google.common.base.Preconditions.
        checkState(Preconditions.java:145)
```

# Guava Collections

# Bi-Map

```
HashBiMap<String, String> englishSpanishMap =
    HashBiMap.<String, String>create();

englishSpanishMap.put("book", "libro");
englishSpanishMap.put("cloud", "nubio");
englishSpanishMap.put("school", "escuela");
englishSpanishMap.put("computer", "ordenador");
```

# Bi-Map

```
HashBiMap<String, String> englishSpanishMap =
    HashBiMap.<String, String>create();

englishSpanishMap.put("book", "libro");
englishSpanishMap.put("cloud", "nubio");
englishSpanishMap.put("school", "escuela");
englishSpanishMap.put("computer", "ordenador");

englishSpanishMap.get("computer") => "ordenador"
```

# Bi-Map

```
HashBiMap<String, String> englishSpanishMap =
    HashBiMap.<String, String>create();

englishSpanishMap.put("book", "libro");
englishSpanishMap.put("cloud", "nubio");
englishSpanishMap.put("school", "escuela");
englishSpanishMap.put("computer", "ordenador");

englishSpanishMap.get("computer") => "ordenador"
englishSpanishMap.put("fill", "llenar");
```

# Bi-Map

```
HashBiMap<String, String> englishSpanishMap =
    HashBiMap.<String, String>create();

englishSpanishMap.put("book", "libro");
englishSpanishMap.put("cloud", "nubio");
englishSpanishMap.put("school", "escuela");
englishSpanishMap.put("computer", "ordenador");

englishSpanishMap.get("computer") => "ordenador"
englishSpanishMap.put("fill", "llenar");
englishSpanishMap.put("feed", "llenar");
```

# Bi-Map

```
HashBiMap<String, String> englishSpanishMap =
    HashBiMap.<String, String>create();

englishSpanishMap.put("book", "libro");
englishSpanishMap.put("cloud", "nubio");
englishSpanishMap.put("school", "escuela");
englishSpanishMap.put("computer", "ordenador");

englishSpanishMap.get("computer") => "ordenador"
englishSpanishMap.put("fill", "llenar");
englishSpanishMap.put("feed", "llenar");
```

IllegalArgumentException

# Bi-Map

```
HashBiMap<String, String> englishSpanishMap =
    HashBiMap.<String, String>create();

englishSpanishMap.put("book", "libro");
englishSpanishMap.put("cloud", "nubio");
englishSpanishMap.put("school", "escuela");
englishSpanishMap.put("computer", "ordenador");

englishSpanishMap.get("computer") => "ordenador"
englishSpanishMap.put("fill", "llenar");
englishSpanishMap.forcePut("feed", "llenar");
```

# Bi-Map

```
HashBiMap<String, String> englishSpanishMap =
   HashBiMap.<String, String>create();

englishSpanishMap.put("book", "libro");
englishSpanishMap.put("cloud", "nubio");
englishSpanishMap.put("school", "escuela");
englishSpanishMap.put("computer", "ordenador");

englishSpanishMap.get("computer") => "ordenador"
englishSpanishMap.put("fill", "llenar");
englishSpanishMap.forcePut("feed", "llenar");

englishSpanishMap.toString() =>
   {computer=ordenador, school=escuela,
    book=libro, cloud=nubio, feed=llenar}
```

# Bi-Map

```
{computer=ordenador, school=escuela,
book=libro, cloud=nubio, feed=llenar}
```

```
BiMap<String, String> spanishEnglishMap =
englishSpanishMap.inverse();
```

```
{escuela=school, nubio=cloud,
ordenador=computer, llenar=feed, libro=book}
```

# Bi-Map

```
{computer=ordenador, school=escuela, book=libro, cloud=nubio,
feed=llenar}
```

```
BiMap<String, String> spanishEnglishMap =
englishSpanishMap.inverse();
```

```
{escuela=school, nubio=cloud, ordenador=computer,
llenar=feed, libro=book}
```

```
spanishEnglishMap.put("futbol", "soccer");
```

```
{escuela=school, nubio=cloud,
 futbol=soccer, ordenador=computer,
 llenar=feed, libro=book}
```

# Bi-Map

```
spanishEnglishMap.toString() =>

{escuela=school, nubio=cloud, futbol=soccer,
ordenador=computer, llenar=feed, libro=book}



englishSpanishMap.toString() =>

{computer=ordenador, school=escuela,
book=libro, cloud=nubio, soccer=futbol
feed=llenar}
```

# Bi-Map

```
BiMap<String, String> spanishEnglishMap =
    englishSpanishMap.inverse();
```

# Multimap

```
ArrayListMultimap<String, Integer>
   superBowlMap =
      ArrayListMultimap.create();
```

Different Flavors:

```
LinkedHashMultimap, LinkedListMultimap,
TreeMultimap, HashMultimap, ListMultimap,
SetMultimap, SortedSetMultimap
```

# Multimap

```
ArrayListMultimap<String, Integer> superBowlMap =
    ArrayListMultimap.create();
superBowlMap.put("Dallas Cowboys", 1972);
superBowlMap.put("Dallas Cowboys", 1993);
superBowlMap.put("Dallas Cowboys", 1994);
superBowlMap.put("Dallas Cowboys", 1996);
superBowlMap.put("Dallas Cowboys", 1978);
superBowlMap.put("Pittsburgh Steelers", 1975);
superBowlMap.put("Pittsburgh Steelers", 1976);
superBowlMap.put("Pittsburgh Steelers", 1979);
superBowlMap.put("Pittsburgh Steelers", 1980);
superBowlMap.put("Pittsburgh Steelers", 2006);
superBowlMap.put("Pittsburgh Steelers", 2009);
```

# Multimap

```
superBowlMap.get("Dallas Cowboys").size() => 5
superBowlMap.get("Pittsburgh Steelers").size() => 6
superBowlMap.get("Buffalo Bills").size() => 0
```

# Multiset a.k.a. Bag

```
Multiset<String> worldCupChampionships =
        HashMultiset.<String>create();
```

Different Flavors:

`EnumMultiset`, `HashMultiset`, `ImmutableMultiset`, `LinkedHashMultiset`, `TreeMultiset`

# Multiset a.k.a. Bag

```
Multiset<String> worldCupChampionships =
    HashMultiset.<String>create();

worldCupChampionships.add("Brazil");
worldCupChampionships.add("Brazil");
worldCupChampionships.add("Brazil");
worldCupChampionships.add("Brazil");
worldCupChampionships.add("Brazil");

worldCupChampionships.add("Italy");
worldCupChampionships.add("Italy");
worldCupChampionships.add("Italy");
worldCupChampionships.add("Italy");
```

# Multiset a.k.a. Bag

```
Multiset<String> worldCupChampionships =
    HashMultiset.<String>create();

worldCupChampionships.add("Brazil");
worldCupChampionships.add("Brazil");
worldCupChampionships.add("Brazil");
worldCupChampionships.add("Brazil");
worldCupChampionships.add("Brazil");

worldCupChampionships.add("Italy");
worldCupChampionships.add("Italy");
worldCupChampionships.add("Italy");
worldCupChampionships.add("Italy");

worldCupChampionships.add("Germany", 3); //explicitly add
                                         //count
```

# Multiset a.k.a. Bag

```
worldCupChampionships.count("Brazil") => 5
worldCupChampionships.count("Italy") => 4
worldCupChampionships.count("Germany") => 3
worldCupChampionships.count("United States") => 0 //Grr!
```

# Immutable vs. Unmodifiable

# Unmodifiability of the JDK

```
Set<Integer> intSet = new HashSet<Integer>();
intSet.add(4);
intSet.add(5);
intSet.add(6);
intSet.add(7);


Set<Integer> unmodifiableSet =
    Collections.unmodifiableSet(intSet);
unmodifiableSet.add(10);
    //UnsupportedOperationException
```

# Unmodifiability of the JDK

```
Set<Integer> intSet = new HashSet<Integer>();
intSet.add(4);
intSet.add(5);
intSet.add(6);
intSet.add(7);

Set<Integer> unmodifiableSet =
    Collections.unmodifiableSet(intSet);
intSet.add(10); // allowed
```

# Unmodifiability of the JDK

```
Set<Integer> intSet = new HashSet<Integer>();
intSet.add(4);
intSet.add(5);
intSet.add(6);
intSet.add(7);

Set<Integer> unmodifiableSet =
    Collections.unmodifiableSet(intSet);
intSet.add(10); // allowed
unmodifiableSet.toString() => [4, 5, 6, 7, 10]
```

# Not Immutable!

You can't modify the collection, but I can!

# Immutability

- All Factory Driven immutable collections for:

maps, multi-sets, multi-maps, sorted sets, sorted maps, lists, sets, and bi-maps

# Immutability with of()

```
Immutable<i>CollectionType</i>.of(E1, E2, E3, E4)
```

# List Immutability

```
ImmutableCollectionType.of(E1, E2, E3, E4)
```

```
List<Integer> integerList =
     ImmutableList.of(4, 4, 5, 6, 7);
```

```
integerList.toString() => [4, 4, 5, 6, 7]
```

# Set Immutability

```
ImmutableCollectionType.of(E1, E2, E3, E4)
```

```
Set<Integer> intSet =
    ImmutableSet.of(6, 7, 7, 8, 9, 10);
```

```
intSet.toString() => [6, 7, 8, 9, 10]
```

# Map Immutability

```
ImmutableCollectionType.of(E1, E2, E3, E4)
```

```
Map<String, String> capitalMap =
      ImmutableMap.of(
         "New Mexico", "Santa Fe",
         "Texas", "Austin",
         "Arizona", "Phoenix");
```

```
capitalMap.toString() =>
     New Mexico -> Santa Fe,
     Texas -> Austin, Arizona -> Phoenix
```

# Bi-Map Immutability

```
ImmutableCollectionType.of(E1, E2, E3, E4)


BiMap<String, String> biMap = ImmutableBiMap.of(
                    "book", "libro",
                    "cloud", "nubio",
                    "school", "escuela",
                    "computer", "ordenador");


biMap.toString() =>
     {book=libro, cloud=nubio, school=escuela,
         computer=ordenador}
```

# Multimap Immutability

```
ImmutableCollectionType.of(E1, E2, E3, E4)
```

```
Multimap<String, Integer> multiMap =
    ImmutableMultimap.of
        ("Dallas Cowboys", 1972,
         "Dallas Cowboys", 1993,
         "Dallas Cowboys", 1994,
         "Dallas Cowboys", 1994,
         "Dallas Cowboys", 1996);
```

# Multimap Immutability

```
ImmutableCollectionType.of(E1, E2, E3, E4)


Multimap<String, Integer> multiMap =
    ImmutableMultimap.of("Dallas Cowboys", 1972,
        "Dallas Cowboys", 1993,
        "Dallas Cowboys", 1994,
        "Dallas Cowboys", 1994,
        "Dallas Cowboys", 1996);
```

But Dallas won in 1978 where is it?
Where are the Steelers information I had earlier?

# The limits of of()

```
Multimap<String, String> multiMap =
    ImmutableMultimap.of(
        "Dallas Cowboys", 1972, "Dallas Cowboys", 1993,
        "Dallas Cowboys", 1994, "Dallas Cowboys", 1994,
        "Dallas Cowboys", 1996, "Dallas Cowboys", 1978,
        "Pittsburgh Steelers", 1975, "Pittsburgh Steelers", 1976,
        "Pittsburgh Steelers", 1979, "Pittsburgh Steelers", 1980,
        "Pittsburgh Steelers", 2006, "Pittsburgh Steelers", 2009);
```

Compile Time Exception: Cannot resolve method

# Immutability with Builders

```
Immutable<i>CollectionType</i>.builder()
```

# List Immutability with Builders

```
ImmutableCollectionType.builder()


List<Integer> intList =
    ImmutableList.<Integer>builder()
        .add(1, 2, 3, 4, 5)
        .addAll(Arrays.asList(6, 7, 8, 9, 10))
        .build();


intList.toString() =>
    [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

# Set Immutability with Builders

```
ImmutableCollectionType.builder()


Set<Integer> intSet =
    ImmutableSet.<Integer>builder()
        .add(1, 2, 3, 4, 5)
        .addAll(Arrays.asList(5, 6, 7, 8, 9, 10))
        .build();


intSet.toString() => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

# Map Immutability With Builders

```
ImmutableCollectionType.builder()


Map<String, String> capitals =
    new ImmutableMap.Builder<String, String>()
        .put("Brazil", "Brasilia")
        .put("United States", "Washington, DC")
        .put("Portugal", "Lisbon")
        .build();


capitals.toString() =>
{Brazil=Brasilia, United States=Washington, DC,
Portugal=Lisbon}
```

# Bi-Map Immutability with Builders

```
ImmutableCollectionType.builder()


BiMap<String, String> biMap = ImmutableBiMap.
      <String, String>builder()
        .put("book", "libro")
        .put("cloud", "nubio")
        .put("school", "escuela")
        .put("computer",
"ordenador").build();


capitals.toString() =>
{Brazil=Brasilia, United States=Washington, DC, Portugal=Lisbon}
```

# Multimap Immutability with Builders

```
ImmutableCollectionType.builder()
Multimap<String, Integer> multiMap =
  ImmutableMultimap.<String, Integer>builder()
    .put("Dallas Cowboys", 1972).put("Dallas Cowboys", 1993)
    .put("Dallas Cowboys", 1994).put("Dallas Cowboys", 1994)
    .put("Dallas Cowboys", 1996).put("Dallas Cowboys", 1978)
    .put("Pittsburgh Steelers", 1975)
    .put("Pittsburgh Steelers", 1976)
    .put("Pittsburgh Steelers", 1979)
    .put("Pittsburgh Steelers", 1980)
    .put("Pittsburgh Steelers", 2006)
    .put("Pittsburgh Steelers", 2009).build();
```

# Multimap Immutability with Builders

```
=> multiMap.toString() => {Dallas Cowboys=[1972, 1993, 1994,
1994, 1996, 1978], Pittsburgh Steelers=[1975, 1976, 1979,
1980, 2006, 2009]}
```

# Predicates

# Predicates

```
Predicate<Integer> isOdd = new Predicate<Integer>(){
      public boolean apply(Integer input) {
        return input % 2 != 0;
      }
};
```

# Predicates

```
Predicate<Integer> isOdd = new Predicate<Integer>(){
     public boolean apply(Integer input) {
       return input % 2 != 0;
     }
};

Collection<Integer> unfiltered =
    Lists<Integer>.newArrayList
      (1, 5, 6, 8, 9, 10, 44, 55, 19);
```

# Predicates

```
Predicate<Integer> isOdd = new Predicate<Integer>(){
     public boolean apply(Integer input) {
       return input % 2 != 0;
     }
};

Collection<Integer> unfiltered =
    Lists<Integer>.newArrayList
      (1, 5, 6, 8, 9, 10, 44, 55, 19);

Collections2.filter(unfiltered, isOdd).toString()
=>[1, 5, 9, 55, 19]
```

# Predicates

```
Predicate<Integer> isOdd = new Predicate<Integer>(){
     public boolean apply(Integer input) {
       return input % 2 != 0;
     }
};

Collection<Integer> unfiltered = Lists<Integer>newArrayList
(1, 5, 6, 8, 9, 10, 44, 55, 19);

Collections2.filter(unfiltered, isOdd).toString()
=> [1, 5, 9, 55, 19]

unfiltered.toString()
=> [1, 5, 6, 8, 9, 10, 44, 55, 19]
```

# Predicate Views

```
Predicate<Integer> isOdd = new Predicate<Integer>() {...};

Collection<Integer> unfiltered = Lists<Integer>newArrayList
(1, 5, 6, 8, 9, 10, 44, 55, 19);

Collection<Integer> filtered = Collections2.filter(unfiltered,
isOdd).toString()
=> [1, 5, 9, 55, 19]

unfiltered.toString()
=> [1, 5, 6, 8, 9, 10, 44, 55, 19]

filtered.add(23); //Good
unfiltered.contains(23) //Yes!
```

# Functions

# Functions

```
Function<Integer, Integer> doubleIt = new
    Function<Integer, Integer>() {
        public Integer apply(Integer from) {
            return from * 2;
        }
    };
```

# Functions

```
Function<Integer, Integer> doubleIt = new
    Function<Integer, Integer>() {
        public Integer apply(Integer from) {
            return from * 2;
        }
    };

Collection<Integer> untransformed = Lists
  .newArrayList
     (1, 5, 6, 8, 9, 10, 44, 55, 19);
```

# Functions

```
Function<Integer, Integer> doubleIt = new
    Function<Integer, Integer>() {
       public Integer apply(Integer from) {
           return from * 2;
       }
    };

Collection<Integer> untransformed = Lists
    .newArrayList(1, 5, 6, 8, 9, 10, 44, 55, 19);

Collections2.transform(untransformed, doubleIt).toString()

=> [2, 10, 12, 16, 18, 20, 88, 110, 38]
```

# Functions

```
Function<Integer, Integer> doubleIt = new
    Function<Integer, Integer>() {
       public Integer apply(Integer from) {
           return from * 2;
       }
    };

Collection<Integer> untransformed = Lists.newArrayList
   (1, 5, 6, 8, 9, 10, 44, 55, 19);

Collections2.transform(untransformed, doubleIt).toString()
   =>[2, 10, 12, 16, 18, 20, 88, 110, 38]

untransformed.toString() => [1, 5, 6, 8, 9, 10, 44, 55, 19]");
```

# Load Cache

# Load Cache

```
LoadingCache<Key, Result> graphs = CacheBuilder.newBuilder()
        .concurrencyLevel(4)
        .maximumSize(10000)
        .expireAfterWrite(10, TimeUnit.MINUTES)
        .expireAfterAccess(10, TimeUnit.MINUTES)
        .initialCapacity(50)
        .weakKeys()
        .weakValues()
        .softValues()
        .removalListener(new MyRemovalListener())
        .build(
                new CacheLoader<Key, Result>() {
                    public Result load(Key key) {
                        return createResult(key);
                    }
                });
```

# Weak Values

- Weak values will be garbage collected once they are weakly reachable.
- Entries with values that have been garbage collected may be counted in `Cache.size()`
- Will never be visible to read or write operations
- `softValues()` is recommended

# Weak Keys

- Weak values will be garbage collected once they are weakly reachable.
- Entries with values that have been garbage collected may be counted in Cache.size()
- Will never be visible to read or write operations;

# Soft Values

- Softly-referenced objects will be garbage-collected in a globally least-recently-used manner, in response to memory demand.
- Better to set a per-cache maximum size instead of using soft references
- Entries with values that have been garbage collected may be counted in `Cache.size()`
- Will never be visible to read or write operations

# Removal Listener

```java
public class MyRemovalListener implements RemovalListener<Key, Result> {

    public void onRemoval(RemovalNotification<Key, Result> keyGraphRemovalNotification) {
        System.out.println(keyGraphRemovalNotification.getCause());
        System.out.println(keyGraphRemovalNotification.getKey());
    }
}
```

# LoadCache Demo

# Utilities

Simple Rule: Use the Plural of the Class for the utility you need.

# Utilities

Simple Rule: Use the Plural of the Class for the utility you need.

`Booleans, Longs, Ints, Floats, Iterables, Iterators, Lists, Longs, Maps, Objects, Multimaps, ObjectArrays, Strings, Shorts, SignedBytes, Sets, Predicates, Multisets, Multimaps, BiMaps, Functions, Bytes`

# Some Favorite Objects Utilities

```
Objects.equal(a,b)
Objects.firstNonNull(a,b)
```

# Some Favorite List Utilities

```
Lists.newArrayList
      ("one, "two", "three")

Lists.newLinkedList(1, 2, 3, 4, 5)

Lists.reverse(someList)

Lists.transform(list, function)
```

# Some Favorite Maps Utilities

```
Maps.newHashMap();
Maps.newEnumMap();
Maps.newLinkedHashMap();
Maps.newConcurrentMap();
Maps.newTreeMap();
Maps.difference(map1,map2).entriesInCommon();
Maps.filterEntries(map, predicate);
Maps.filterKeys(map, predicate);
Maps.filterValues(map, predicate);
Maps.transformEntries(map, transformer);
Maps.transformValues(map, function);
```

# Finding Differences

```
Map<String, String> stateCaps =
     ImmutableMap.<String, String>builder()
         .put("Tallahassee", "Florida")
         .put("Santa Fe", "New Mexico")
         .put("Trenton", "New Jersey")
         .put("Olympia", "Washington")
         .put("Albany", "New York").build();
Map<String, String> stateCaps2 =
     ImmutableMap.<String, String>builder()
         .put("Tallahassee", "Florida")
         .put("Raleigh", "North Carolina")
         .put("Bismarck", "North Dakota").build();
MapDifference<String, String> diff =
     Maps.difference(stateCaps, stateCaps2);
diff.entriesOnlyOnLeft().size() // 4
diff.entriesOnlyOnRight().size() // 2
```

# Finding Common Entries

```
Map<String, String> stateCaps =
     ImmutableMap.<String, String>builder()
        .put("Tallahassee", "Florida")
        .put("Santa Fe", "New Mexico")
        .put("Trenton", "New Jersey")
        .put("Olympia", "Washington")
        .put("Albany", "New York").build();
Map<String, String> stateCaps2 =
     ImmutableMap.<String, String>builder()
        .put("Tallahassee", "Florida")
        .put("Raleigh", "North Carolina")
        .put("Bismarck", "North Dakota").build();
Map<String, String> common = Maps.difference(stateCaps,
    stateCaps2).entriesInCommon();
common.size() // 1
common.get("Tallahassee") //"Florida
```

# Using Predicate, FilteredValuesUtilities

```
Map<String, String> stateCaps =
    ImmutableMap.<String,String>builder()
    .put("Tallahassee", "Florida")
    .put("Santa Fe", "New Mexico")
    .put("Trenton", "New Jersey")
    .put("Olympia", "Washington")
    .put("Albany", "New York").build();
Predicate<CharSequence> startsWithNew =
     Predicates.containsPattern("New.*");
Map<String, String> filtered =
      Maps.filterValues(stateCaps,startsWithNew);
filtered.size(); //3
```

# Some Favorite Iterables Utilities

```
Iterables.concat(list1, list2);
Iterables.elementsEqual(list1, list2);
Iterables.cycle(list);
Iterables.filter(list, clazz);
Iterables.filter(list, predicate);
Iterables.partition(list, size);
Iterables.paddedPartition(list, size);
Iterables.transform(list, function);
Iterables.tryFind(list, predicate);
```

# Using cycle

```
List<Integer> list =
  Lists.newArrayList(1, 2, 3, 4, 5);
Iterable iterable =
  Iterables.cycle(list);
Iterator it = iterable.iterator();
for (int i = 0; i < 1000; i++){
  it.next();
} // 1
```

# Using partition

```
List<Integer> list =
   Lists.newArrayList(1, 2, 3, 4, 5);
Iterable iterable =
   Iterables.partition(list, 2);
Iterator it = iterable.iterator();
it.next(); //List(1, 2)
it.next(); //List(3, 4)
it.next(); //List(5);
```

# Using padded partition

```
List<Integer> list =
   Lists.newArrayList(1, 2, 3, 4, 5);
Iterable iterable =
   Iterables.paddedPartition(list, 2);
Iterator it = iterable.iterator();
it.next(); //List(1, 2)
it.next(); //List(3, 4)
it.next(); //List(5, null)
```

# Some Favorite Strings Utilities

```
Strings.isNullOrEmpty(string)
Strings.nullToEmpty(string)
Strings.padEnd(string, minLength, char)
Strings.padEnd(string, minLength, char)
Strings.padStart(string, minLength, char)
Strings.repeat(string, times)
```

# Moral of the Story

If it feels like someone else has already developed what you are trying to, do look it up.

# Ordering

```
public class StarWarsEpisode {
    private String name;
    private int number;
    private int year;

    //getters, toString, hashCode, equals
}
```

# Ordering

```
public class StarWarsCharacter implements
     Comparable<StarWarsCharacter> {
        private String name;
        private StarWarsEpisode firstAppearance;


        //getters, toString, hashCode, equals

        public int compareTo(StarWarsCharacter o) {
            return this.name.compareTo(o.name) +
                    this.firstAppearance.getYear() -
                    o.firstAppearance.getYear();
    }
}
```

# Ordering

```
aNewHope = new StarWarsEpisode
          ("A New Hope", 4, 1977);
empireStrikesBack = new StarWarsEpisode
          ("The Empire Strikes Back", 5, 1980);
returnOfTheJedi = new StarWarsEpisode
          ("Return Of The Jedi", 6, 1983);
phantomMenace = new StarWarsEpisode
          ("The Phantom Menace", 1, 1999);
attackOfTheClones = new StarWarsEpisode
          ("Attack Of The Clones", 2, 2002);
revengeOfTheSith = new StarWarsEpisode
          ("Revenge Of The Sith", 3, 2005);
```

# Ordering

```
hanSolo = new StarWarsCharacter
          ("Han Solo", aNewHope);
lukeSkywalker = new StarWarsCharacter
          ("Luke Skywalker", aNewHope);
princessLeia = new StarWarsCharacter
          ("Princess Leia", aNewHope);
landoCalrissian = new StarWarsCharacter
          ("Lando Calrissian", empireStrikesBack);
bobaFett = new StarWarsCharacter
          ("Boba Fett", empireStrikesBack);
```

# Ordering

```
public class StarWarsEpisodeYearComparator
    implements Comparator<StarWarsEpisode> {
        public int compare (StarWarsEpisode o1,
                            StarWarsEpisode o2) {
            return o1.getYear() - o2.getYear();
     }
}
```

# Ordering

```
Ordering.from(
      new StarWarsEpisodeYearComparator())
          .max(aNewHope, phantomMenace)


=> phantomMenace
```

# Ordering

```
public class StarWarsCharacterNameComparator implements
    Comparator<StarWarsCharacter> {
        public int compare(StarWarsCharacter o1,
                           StarWarsCharacter o2) {
            return o1.getName().compareTo(o2.getName());
        }
}
```

# Ordering

```
Ordering.from(new StarWarsCharacterYearComparator())
    .compound(new StarWarsCharacterNameComparator())
    .sortedCopy(Lists.newArrayList(
                bobaFett,princessLeia,
                landoCalrissian, lukeSkywalker,
                hanSolo)).toString();
```

```
=> [Han Solo, Luke Skywalker, Princess Leia, Boba
Fett, Lando Calrissian]
```

# Ordering

```
Ordering<String> byLengthOrdering =
        new Ordering<String>() {
             public int compare(String left, String right) {
                 return (left.length() - right.length());
             }
         };

byLengthOrdering.max(hanSolo.getName(),
                     lukeSkywalker.getName(),
                     princessLeia.getName())

=> "Luke Skywalker"
```

# Ordering

```
Ordering.explicit(phantomMenace,
    attackOfTheClones, revengeOfTheSith,
    returnOfTheJedi, aNewHope,
    empireStrikesBack).max
(revengeOfTheSith, aNewHope)

=> aNewHope
```

# Ordering

```
byLengthOrdering = new Ordering<String>() {
            public int compare(String left, String right) {
                return (left.length() - right.length());
            }
        };


byLengthOrdering.nullsLast()
       .sortedCopy(Arrays.asList(hanSolo.getName(), null,
            lukeSkywalker.getName(), null,
                princessLeia.getName())).toString() =>


"[Han Solo, Princess Leia, Luke Skywalker, null, null]"
```

# Ordering

```
byLengthOrdering = new Ordering<String>() {
            public int compare(String left, String right) {
                return (left.length() - right.length());
            }
};


byLengthOrdering.isOrdered
        (Arrays.asList(hanSolo.getName(),
                       princessLeia.getName(),
                       lukeSkywalker.getName(),
                       lukeSkywalker.getName())
=> true
```

# Ordering

```
byLengthOrdering = new Ordering<String>() {
            public int compare(String left, String right) {
                return (left.length() - right.length());
            }
};


byLengthOrdering.isStrictlyOrdered
        (Arrays.asList(hanSolo.getName(),
                       princessLeia.getName(),
                       lukeSkywalker.getName(),
                       lukeSkywalker.getName())
=> false
```

# Ordering

```
StarWarsCharacterNameComparator
    starWarsCharacterNameComparator = new
        StarWarsCharacterNameComparator();

StarWarsCharacter key = new
       StarWarsCharacter("Princess Leia", null);

Ordering.from(starWarsCharacterNameComparator)
    .binarySearch(Arrays.asList(bobaFett, hanSolo,
                  landoCalrissian, lukeSkywalker,
                  princessLeia), key)
=> 4
```

# Ordering

```
public class StarWarsCharacter implements
     Comparable<StarWarsCharacter> {
        private String name;
        private StarWarsEpisode firstAppearance;


        //getters, toString, hashCode, equals

        public int compareTo(StarWarsCharacter o) {
            return this.name.compareTo(o.name) +
                    this.firstAppearance.getYear() -
                    o.firstAppearance.getYear();
    }
}
```

# Ordering

```
Ordering.natural()
      .sortedCopy(Arrays.asList
          (bobaFett, hanSolo, lukeSkywalker,
             princessLeia, landoCalrissian)).toString()


=>"[Boba Fett, Han Solo, Lando Calrissian, Luke
Skywalker, Princess Leia]"
```

# Optional

“I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.”

- Sir Charles Antony Richard Hoare

# Optional

```
public Optional<String> getMiddleName(String fullName) {
     String[] parts = fullName.split(" ");
     if (parts.length <= 2) return Optional.absent();
     return Optional.of(parts[1]);
}

getMiddleName("Marge Simpson")); //Optional.absent()
getMiddleName
        ("Homer J. Simpson"));//Optional.of("J."));
```

# Optional

```
 public Optional<String> getValueFromInternalMap(String key) {
    Map<String, String> maps = ImmutableMap.of("One", "1", "Two", "2", "Three", "3");
    return Optional.fromNullable(maps.get(key));
}
getValueFromInternalMap("Nine")); // Optional.absent()
getValueFromInternalMap("One").get();  // "One"
getValueFromInternalMap("One").isPresent(); // true
```

# EventStream

- Dispatches Events
- Easier than the `java.util.Observer` and `java.util.Observable`
- Requires the components to explicitly register with one another
- Posters, Handlers, Dead Events

# Broadcast Event

```
public class BroadcastEvent {
     private String message;

     public BroadcastEvent(String message) {
        this.message = message;
     }

     public String getMessage() {
        return message;
     }

     //equals, hashcode, toString
  }
```

# Broadcaster

```
public class Broadcaster {
      private EventBus eventBus;

      public void setEventBus(EventBus eventBus) {
         this.eventBus = eventBus;
      }

      public void broadcastToAll() {
         this.eventBus.post(
         new BroadcastEvent("The Guava Revolution
             will not be televised"));
      }
}
```

# Broadcast Event

```
public class BroadcastEvent {
     private String message;

     public BroadcastEvent(String message) {
        this.message = message;
     }

     public String getMessage() {
        return message;
     }

     //equals, hashcode, toString
  }
```

# Subscriber

```
public class Subscriber {
     private List<String> messages =
          Lists.newArrayList();

     @Subscribe
     public void eventOccured(BroadcastEvent event) {
        messages.add(event.getMessage());
     }

     public int getCount() {
        return messages.size();
     }

     public List<String> getMessages() {
        return ImmutableList.copyOf(messages);
     }
  }
```

# Using the EventBus

```
EventBus eventBus = new EventBus();
Subscriber subscriber = new Subscriber();
eventBus.register(subscriber);

Broadcaster broadcaster = new Broadcaster();
broadcaster.setEventBus(eventBus);

broadcaster.broadcastToAll();
broadcaster.broadcastToAll();
broadcaster.broadcastToAll();

subscriber.getCount() // 3
```

# Questions?

# Thanks!

Email: dhinojosa@evolutionnext.com

Twitter: @dhinojosa

Google Plus: gplus.to/dhinojosa

Linked In: www.linkedin.com/in/dhevolutionnext