Break New Ground

San Francisco
September 16–19, 2019

Collections Corner Cases

Stuart W. Marks

Consulting Member of Technical Staff
Java Platform Group
Oracle

Twitter: @stuartmarks

Stack Overflow:

Stuart Marks
90,428
●29 ●152 ●215

#CollectionsCornerCases

Overview

- Two corners of the Collections Framework API
    - Not new, they've always been there
    - Not well known, and even obscure
- Part 1: View Collections
- Part 2: Sorted Collections
    - … and Comparators and "consistent with equals"

View Collections

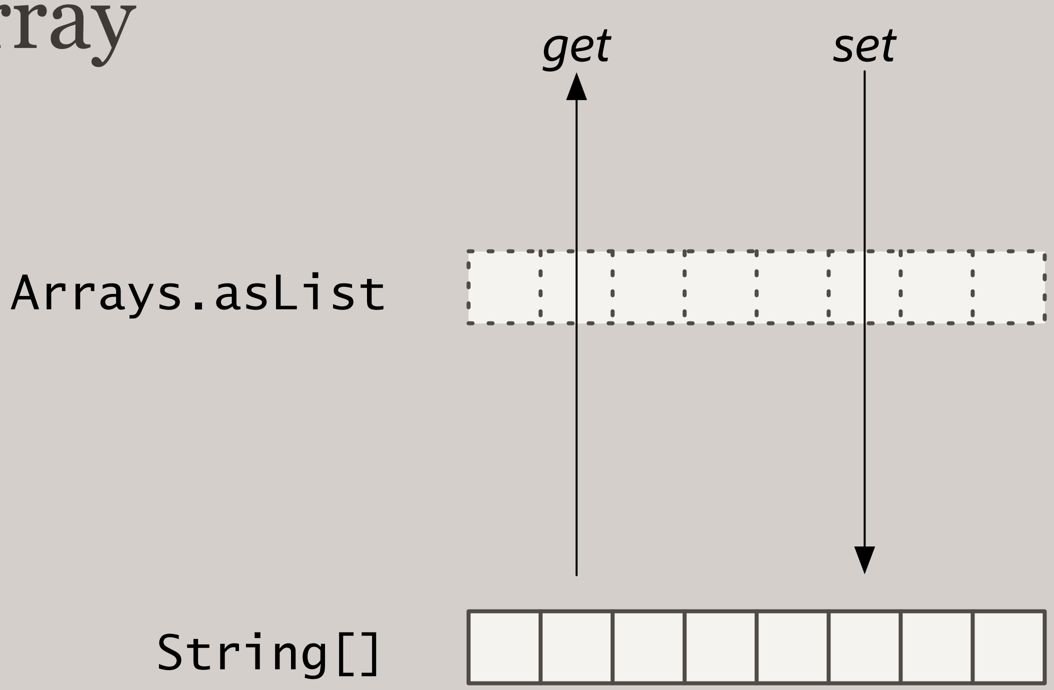
- Most Collections contain their own elements
  - ArrayList, HashMap, etc.
- View Collections don't contain their own elements
  - instead, the elements are stored elsewhere

List View of Array

```
String[] array = { "a", "b", "c", "d", "e", "f" };

List<String> list = Arrays.asList(array);

array[0] = "XXX";
    // list is now [XXX, b, c, d, e, f]

list.set(1, "YYY");
    // array is now { XXX, YYY, c, d, e, f }
```

#CollectionsCornerCases

List View of Array



Map Views

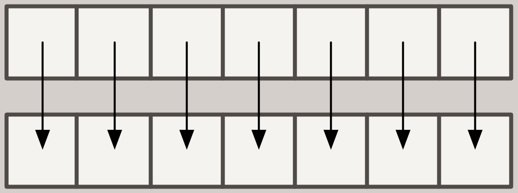
```
Map<Integer, String> map = new HashMap<>(...);

Set<Integer> keySet = map.keySet();

Collection<String> values = map.values();

Set<Map.Entry<Integer, String>> entrySet =
    map.entrySet();
```

Map.Entry

```
interface Map.Entry<K, V> {
    K getKey();
    V getValue();
    V setValue(V value);
}
```

A HashMap

`HashMap<Integer, String>`

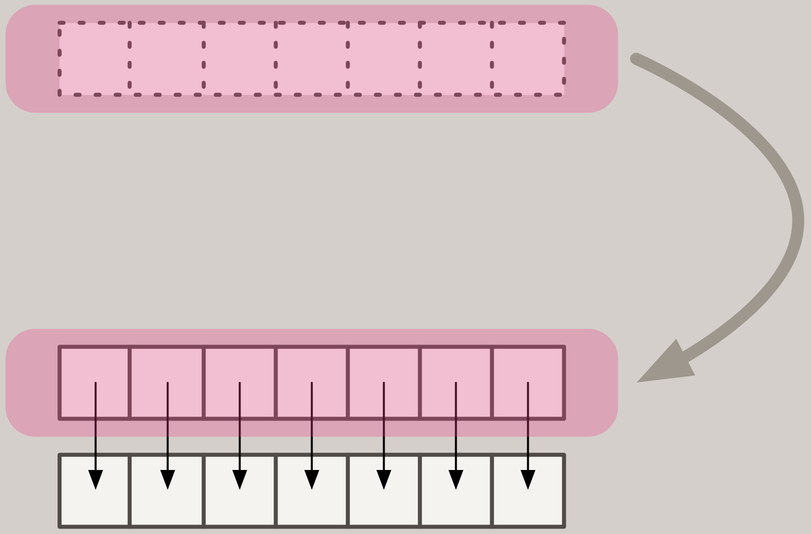


#CollectionsCornerCases

Map Key Set View

```
Set<Integer> Map.keySet()
```

```
HashMap<Integer, String>
```

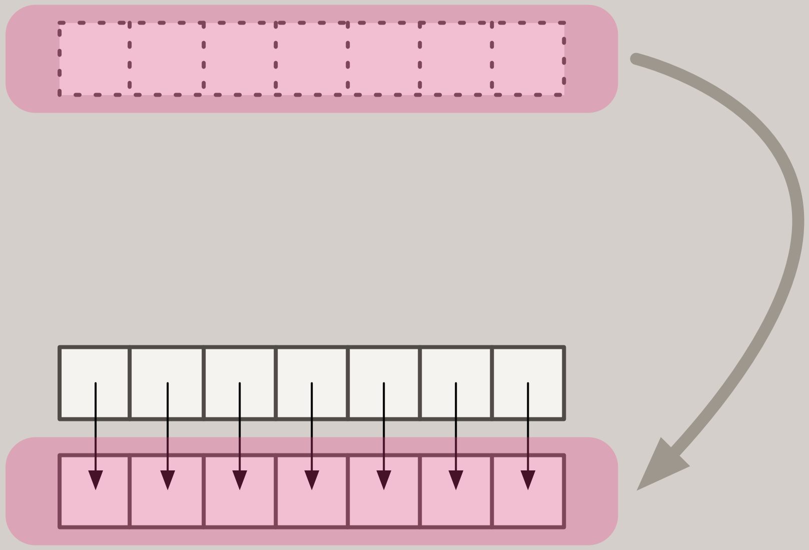


#CollectionsCornerCases

Map Values View

`Collection<String> Map.values()`

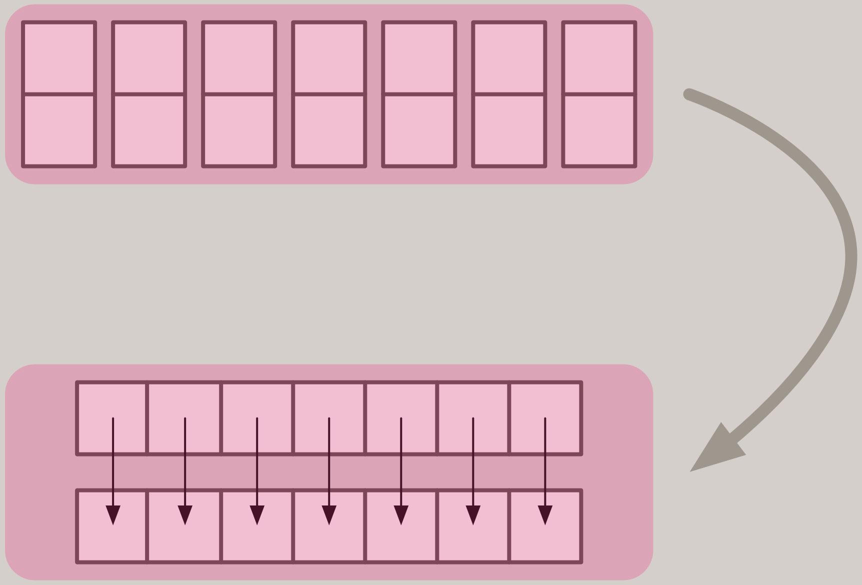
`HashMap<Integer, String>`



Map Entry-set View

```
Set<Map.Entry<Integer, String>>
                   Map.entrySet()
```

```
HashMap<Integer, String>
```



DEMO

#CollectionsCornerCases

Extended Map Views Example

- Given two maps mapA and mapB, find the keys of mapB corresponding to values that occur in both maps

```
var mapA = Map.of(1, "a", 2, "b", 3, "c")
var mapB = Map.of(5, "a", 6, "d", 7, "c")
```

- Result should be a set containing…

```
[5, 7]
```

- Maps might contain 15,000 entries
  - algorithm is important!
  - https://stackoverflow.com/a/57931314/1441122

Solution Attempt #1

```
Map<Integer, String> mapA = ... ;
Map<Integer, String> mapB = ... ;

Set<Integer> result = mapB.keySet().stream()
  .filter(keyB -> mapA.keySet().stream()
      .filter(keyA ->
              mapA.get(keyA).equals(mapB.get(keyB)))
      .count() > 0)
  .collect(toSet());
```

Solution Attempt #2

```
Map<Integer, String> mapA = ... ;
Map<Integer, String> mapB = ... ;

Set<Integer> result = mapB.entrySet().stream()
    .filter(entry ->
            mapA.values().contains(entry.getValue()))
    .map(Map.Entry::getKey)
    .collect(toSet());
```

Insight

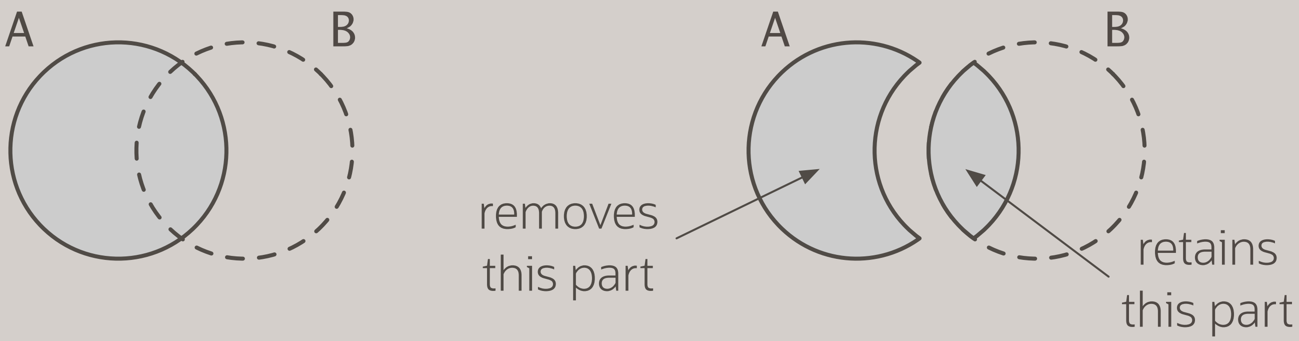
- Use HashSet to strength-reduce contains() operations
  - contains() on the values() view is linear, O(n)
  - doing it for every key results in O(n^2)
  - bring it into a HashSet to reduce contains() to constant, O(1)
  - overall time reduced from O(n^2) to O(n)

Solution Attempt #2 + variation

```
Map<Integer, String> mapA = ... ;
Map<Integer, String> mapB = ... ;

var aVals = new HashSet<>(mapA.values());
Set<Integer> result = mapB.entrySet().stream()
   .filter(entry -> aVals.contains(entry.getValue()))
   .map(Map.Entry::getKey)
   .collect(toSet());
```

Insight+

- Consider set intersection
  - problem statement says "… values that occur in both maps"
- Set intersection achieved using the retainAll() method
  - A.retainAll(B) leaves set intersection in A



Insight++

- Use bulk operations on the values() view of the map
  - removes unwanted entries from the map
  - leaving only the desired entries
- Use keySet() view
  - we don’t want the whole map, just the keys

Solution Attempt #3

```
Map<Integer, String> mapA = ... ;
Map<Integer, String> mapB = ... ;

var aVals = new HashSet<>(mapA.values());
var mapBcopy = new HashMap<>(mapB);
mapBcopy.values().retainAll(aVals);
Set<Integer> result = mapBcopy.keySet();
```

Part 1 Summary

- Know and use view collections
  - view collections of Map are particularly interesting
- Collections have rich set of bulk operations
  - combined with view collections can be very powerful
  - sometimes can be more effective than streams

Part 2: Sorted Collections & Comparators

- Sorted Collections
  - interfaces
    - SortedSet, SortedMap, NavigableSet, NavigableMap
  - implementations
    - TreeSet, TreeMap, ConcurrentSkipListSet, ConcurrentSkipListMap
  - I'll talk about SortedSet but same issues apply to all
- Sorted Collections are ordered by a *Comparator*
  - also Comparable and "natural order" but issues are the same
  - a Comparator might or might not be "consistent with equals"

Iteration Order

- For HashSet and Set.of(), iteration order is *undefined*

```
Set.of("a", "bb", "ccc", "dddd", "eeeee")
[eeeee, bb, a, ccc, dddd]

new HashSet<>(List.of("a", "bb", "ccc", "dddd", "eeeee"))
[bb, a, ccc, eeeee, dddd]
```

- For SortedSet et. al., iteration order is *well-defined*
  - you provide the order using a Comparator
  - (“comparison method”)

Comparator<T>

- Interface with one method: `int compare(T a, T b)`
- Compares two objects and reports their relative ordering

< 0	"a is less than b"
0	"a is equal to b"
> 0	"a is greater than b"

- Must impose a *total ordering*
- Must follow rules: transitive, reflexive, antisymmetric, ...
- Example: string alphabetical order (case sensitive)

DEMO

Set Membership

- Typical set (e.g., HashSet)
  - contains no two elements (a, b) where a.equals(b) is true
- Sorted set (e.g., TreeSet)
  - contains no two elements (a, b) where compare(a, b) == 0
- In other words, the comparator not only determines *order* but also *membership*

Comparator<T>

- Interface with one method: `int compare(T a, T b)`
- Compares two objects and reports their relative ordering

< 0	“a is less than b”
0	“~~a is equal to b~~” $\longrightarrow$ “a is *equivalent* to b in the context of this ordering”
> 0	“a is greater than b”

- Equivalence not a property of the objects themselves (like equals) but instead determined by comparator

“Consistent With Equals”

From the Comparator specification:

> *The ordering imposed by a comparator* **c** *on a set of elements* **S** *is said to be consistent with equals if and only if* **c.compare(e1, e2)==0** *has the same boolean value as* **e1.equals(e2)** *for every* **e1** *and* **e2** *in* **S***.*
>
> *Caution should be exercised when using a comparator capable of imposing an ordering inconsistent with equals to order a sorted set (or sorted map). Suppose a sorted set (or sorted map) with an explicit comparator* **c** *is used with elements (or keys) drawn from a set* **S***. If the ordering imposed by* **c** *on* **S** *is inconsistent with equals, the sorted set (or sorted map) will behave “strangely.” In particular the sorted set (or sorted map) will violate the general contract for set (or map), which is defined in terms of* **equals***.*

“Consistent With Equals”

From the SortedSet specification:

> *Note that the ordering maintained by a sorted set (whether or not an explicit comparator is provided) must be consistent with equals if the sorted set is to correctly implement the Set interface. (See the Comparable interface or Comparator interface for a precise definition of consistent with equals.) This is so because the Set interface is defined in terms of the* **equals** *operation, but a sorted set performs all element comparisons using its* **compareTo** *(or* **compare***) method, so two elements that are deemed equal by this method are, from the standpoint of the sorted set, equal. The behavior of a sorted set is well-defined even if its ordering is inconsistent with equals; it just fails to obey the general contract of the Set interface.*

WAT

DEMO

Explanation

- Membership contract
  - Typical set membership determined by equals()
  - SortedSet membership determined by comparator
- If these disagree (“inconsistent with equals”)
  - “behaves strangely”
  - “violates the general contract of Set”
- Behavior can be unexpected if you’re expecting one set’s semantics but the other’s is used

Practical Impact

- Obscure edge case, never occurs in real world?
  - actually occurs quite easily
  - many comparators for sorting are inconsistent with equals
- Repurposing a sorting-comparator for a sorted set can be perilous
  - “duplicate” elements will be eliminated

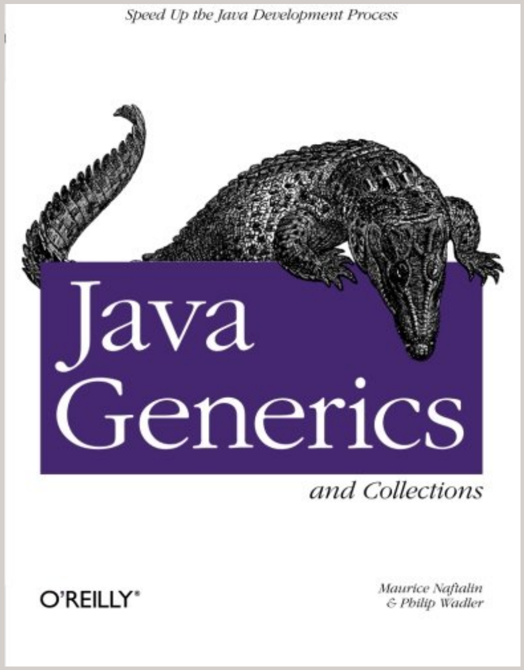
DEMO

Lessons

- SortedSet with inconsistent-with-equals comparator
  - use only with SortedSets with the *same* comparator
  - DO NOT MIX with sets that use other semantics
- IdentityHashMap keys
  - membership determined by == , inconsistent with equals
- Surprisingly easy to create comparators that are inconsistent with equals
- When in doubt, put elements in a list and sort them

Summary

- Don’t forget about good old collections
  - especially Map views and bulk operations
  - sorted collections very powerful
    - can be used safely in isolation
    - dangerous if mixed without care
- Following up
  - Q&A: Oracle Groundbreaker Hub, Wed 1:30pm
  - Naftalin and Wadler, *Java Generics and Collections*
  - Twitter: #CollectionsCornerCases


Speed Up the Java Development Process
Java Generics
and Collections
O'REILLY
Maurice Naftalin & Philip Wadler

Safe Harbor

ORACLE CODE ONE

Break New Ground

San Francisco
September 16–19, 2019

- Map Views Demo
  - jshell overview
    - introduced in Java 9
    - evaluates expressions and prints the result; REPL: read-eval-print loop
    - (mostly) no semicolons, TAB completion
    - jshell commands begin with /

      -
        ```
        /open ccc.jshell
        ```
      - loads some stuff from a file

      -
        ```
        1 + 2
        "hello".toUpperCase()
        var list = List.of(1, 2, 3)
        /var list
        ```
      - it's a bit unfortunate that "var" is part of the language but "slash var" is a jshell command the displays a variable's type and value
  - these are views onto the map, so changes to the map affect the views, and changes from the view affect the underlying map

  -
    ```
    map
    {1=a, 2=b, 3=c, 4=d, 5=e}
    var keySet = map.keySet()
    var values = map.values()
    var entrySet = map.entrySet()
    map.put(6, "f")
    // show map, keySet, and values
    keySet.remove(3)
    // show keySet, map, and values
    keySet.removeIf(k -> k % 2 == 0)
    // show keySet, map, and values
    // can't add to keySet, values, or entrySet; throws UOE
    ```
  - What's the point of entrySet? You can iterate set of entries and remove them using an iterator, or call setValue on an entry

  -
    ```
    var it = entrySet.iterator()
    var entry = it.next()
    entry.getKey()
    entry.getValue()
    entry.setValue("XXX")
    // entry, entrySet, and map have all changed
    ```

▼ Sorted Collections and Comparators

- Simple Comparator example; alpha order, case sensitive

-
  ```
  Comparator<String> cso = String::compareTo
  cso.compare("a", "z")
  cso.compare("z", "a")
  cso.compare("x", "x")
  ```
- Use comparator with a sorted collection

-
  ```
  strings
  ["eeeee", "bb", "a", "ccc", "dddd"] // random
  var tree = new TreeSet<>(cso)
  tree.addAll(strings) // show sorted
  tree.add("cz")
  tree.add("TTT")
  ```
- A sorted collection is kind of like a sorted list, right? No: treatment of duplicates

-
  ```
  var list = new ArrayList<>(strings)
  list.add("bb")
  list.sort(cso)
  // duplicates preserved
  tree.clear()
  tree.addAll(list)
  // duplicates elided!
  ```

▼ Inconsistent with Equals Demo

-
  ```
  cso.compare("a", "a") // 0
  "a".equals("a") // true
  cso.compare("a", "A") // nonzero
  "a".equals("A") // false
  // what could be inconsistent? if == 0 then equals ought to be true right?
  Comparator<String> cio = String.CASE_INSENSITIVE_ORDER
  cio.compare("a", "a") // 0
  cio.compare("a", "A") // 0
  "a".equals("A") // false
  ```
- Being inconsistent with equals is NOT A BUG; there are valid use cases
- They can give rise to inconsistency, "strange behavior", "violate general set contract"

▼ Sorted Collection with I-W-E Comparator

  -
    ```
    var tree = new TreeSet<>(cio)
    tree.addAll(List.of("MMM", "aaa", "zzz")
    // MMM sorts in middle; case insensitive
    tree.contains("MMM") // true
    tree.contains("mmm") // also true
    ```

  -
    ```
    var hash = new HashSet<>(tree)
    hash.contains("MMM") // true
    hash.contains("mmm") // false
    ```
  - set1.equals(set2) is true iff sizes equal and every element of set2 is also contained in set1

  -
    ```
    // show hash and tree; they look equal
    hash.equals(tree) // true
    tree.equals(hash) // true
    ```

  -
    ```
    hash.remove("MMM")
    hash.add("mmm")
    // show hash and tree; they look unequal
    hash.equals(tree) // false
    tree.equals(hash) // true!!
    ```

- Employee Demo

  -
    ```
    /list Employee
    /var employees
    employees.forEach(System.out::println)
    Comparator<Employee> salaryOrder = Comparator.comparing(Employee::getSalary)
    employees.sort(salaryOrder)
    ```
  - works fine for sorting a list
  - suppose your manager asks you to maintain a data structure in the same order; easy, right?

  -
    ```
    var tree = new TreeSet<>(salaryOrder)
    tree.addAll(employees)
    tree
    ```
  - Pat is missing! Why? salaryOrder is inconsistent with equals

  -
    ```
    pat.equals(terry) // false
    salaryOrder.compare(pat, terry) // 0
    ```
  - it considers unequal employees (pat and terry) that have the same salary to be equal thus duplicates
  - solution
    - either augment the comparator to distinguish EVERY object (e.g., employee ID)
    - or keep in list and sort, which preserves "duplicates"

```
// ========== MAPS ==========

var origMap = Map.of(1, "a", 2, "b", 3, "c", 4, "d", 5, "e")
var map = new HashMap<>(origMap)
var map2 = Map.of(1, "X", 2, "B", 3, "C", 4, "D", 5, "E")
var mapA = Map.of(1, "a", 2, "b", 3, "c")
var mapB = Map.of(5, "a", 6, "d", 7, "c")

// ========== COMPARATORS ==========

var strings = new ArrayList<>(List.of("a", "bb", "ccc", "dddd",
"eeeee"))
Collections.shuffle(strings)

// ========== EMPLOYEE ==========

public class Employee {
    final String name;
    final int salary;

    public Employee(String name, int salary) {
        this.name = name;
        this.salary = salary;
    }

    public String toString() {
        return String.format("Employee[%s,%s %d]", name,
pad(name, 5), salary);
    }

    public String getName() { return name; }

    public int getSalary() { return salary; }

    public boolean equals(Object o) {
        if (o == this) return true;
        if (! (o instanceof Employee)) return false;
        Employee other = (Employee) o;
        return this.name.equals(other.name) && (this.salary ==
other.salary);
    }

    public int hashCode() {
        return (name.hashCode() * 31) ^ salary;
    }

    private static String pad(String s, int width) {
        return " ".repeat(Math.max(width - s.length(), 0));
    }
}

Employee pat;
Employee terry;

var origEmployees = List.of(
    new Employee("Robin", 400),
    terry = new Employee("Terry", 300),
    new Employee("Dana", 100),
    new Employee("Kelly", 200),
    pat = new Employee("Pat", 300),
    new Employee("Glenn", 500)
);

var employees = new ArrayList<>(origEmployees);

System.out.println("ccc loaded")
```