Working with Collections

Jim Wilson
MOBILE SOLUTIONS DEVELOPER & ARCHITECT
@hedgehogjim blog.jwhh.com

Overview

The role of collections

Collections and type safety

Common collection methods

Collections and entry equality

New collection methods in Java 8

Converting between collections and arrays

Common collection interfaces and classes

Sorting behavior

Map collections

Managing Groups of Data

Apps often need to manage data in commonly typed groups

- Most basic solution is to use arrays

Arrays have limitations

- Statically sized
- Requires explicit position management
- Little more than a bunch of values

Collections provide more powerful options

The Role of Collections

Collections hold and organize values

- Iterable
- Can provide type safety
- Tend to dynamically size

A wide variety of collections are available

- May be a simple list of values
- Can provide optimization or sophistication
  - Ordering
  - Prevent duplicates
  - Manage data as name/value pairs

A Simple Collection of Objects

```
ArrayList list = new ArrayList();

list.add(“Foo”);
list.add(“Bar”);

System.out.println(“Elements: ” + list.size());

for(Object o:list)
  System.out.println(o.toString());

String s = (String)list.get(0);

SomeClassIMadeUp c = new SomeClassIMadeUp();
list.add(c);
```

Collections and Type Safety

By default collections hold Object types

- Must convert return values to desired type
- Doesn't restrict types of values added

Collections can be type restricted

- Uses the Java concept of generics
- Type specified during collection creation

Collection type restriction is pervasive

- Return values appropriately typed
- Adding values limited to appropriate type

A Strongly Typed Collection

```
ArrayList<String> list = new ArrayList< >();

list.add(“Foo”);
list.add(“Bar”);

System.out.println(“Elements: ” + list.size());

for(String o:list)
  System.out.println(o );

String s = list.get(0);

SomeClassIMadeUp c = new SomeClassIMadeUp();
list.add(c);
```

Collection Interface

Each collection type has its own features

- But there are many that are common

Collection interface

- Provides common collection methods
- Implemented by most collection types
  - Map collections are notable exception
- Extends Iterable interface

Common Collection Methods

Method	Description
size	Returns number of elements
clear	Removes all elements
isEmpty	Returns true if no elements
add	Add a single element
addAll	Add all members of another collection

Adding Members from Another Collection

```
ArrayList<String> list1 = new ArrayList<>();
list1.add("Foo");
list1.add("Bar");

LinkedList<String> list2 = new LinkedList<>();
list2.add("Baz");
list2.add("Boo");

list1.addAll(list2);

for(String s:list1)
  System.out.println(s);
```

Does not affect list2

```
Foo
Bar
Baz
Boo
```

Common Equality-based Methods

Method	Description
contains	Return true if contains element
containsAll	Return true if contains all members of another collection
remove	Remove element
removeAll	Remove all elements contained in another collection
retainAll	Remove all elements not contained in another collection

Tests all use the equals method

Removing a Member

```
public class MyClass {
  String label, value; // getters elided for clarity

  public MyClass(String label, String value) {
    // assign label & value to member fields
  }

  public boolean equals(Object o) {
    MyClass other = (MyClass) o;
    return value.equalsIgnoreCase(other.value);
  }
}
```

Removing a Member

```
ArrayList<MyClass> list = new ArrayList<>();

MyClass v1 = new MyClass("v1", "abc");
MyClass v2 = new MyClass("v2", "abc");
MyClass v3 = new MyClass("v3", "abc");

list.add(v1);
list.add(v2);
list.add(v3);

list.remove(v3);

for(MyClass m:list)
  System.out.println(m.getLabel());
```

Uses equals method to find match

```
v2
v3
```

Java 8 Collection Methods

Java 8 introduced lambda expressions

- Simplify passing code as arguments

Collection methods that leverage lambdas

- forEach
  - Perform code for each member
- removeIf
  - Remove element if test is true

Using forEach Method

```
ArrayList<MyClass> list = new ArrayList<>();

MyClass v1 = new MyClass("v1", "abc");
MyClass v2 = new MyClass("v2", “xyz");
MyClass v3 = new MyClass("v3", "abc");

list.add(v1);
list.add(v2);
list.add(v3);

list.forEach(m -> System.out.println(m.getLabel()));
```

```
v1
v2
v3
```

Using removeIf Method

```
ArrayList<MyClass> list = new ArrayList<>();

MyClass v1 = new MyClass("v1", "abc");
MyClass v2 = new MyClass("v2", “xyz");
MyClass v3 = new MyClass("v3", "abc");

list.add(v1);
list.add(v2);
list.add(v3);

list.removeIf(m -> m.getValue().equals(“abc”));

list.forEach(m -> System.out.println(m.getLabel()));
```

v2

Converting Between Collections and Arrays

Sometimes APIs require an array

- Often due to legacy or library code

Collection interface can return an array

- toArray() method
  - Returns Object array
- toArray(T[] array) method
  - Returns array of type T

Array content can be retrieved as collection

- Use Arrays class' asList method

Retrieving an Array

```
ArrayList<MyClass> list = new ArrayList<>();
list.add(new MyClass("v1", "abc"));
list.add(new MyClass("v2", “xyz"));
list.add(new MyClass("v3", "abc"));

Object[] objArray = list.toArray();

MyClass[] a1 = list.toArray(new MyClass[0]);

MyClass[] a2 = new MyClass[3];
MyClass[] a3 = list.toArray(a2);

if(a2 == a3)
  System.out.println(“a2 & a3 reference the same array”);
```

Retrieving a Collection from an Array

```
MyClass[] myArray= {
  new MyClass("val1", "abc"),
  new MyClass("val2", “xyz"),
  new MyClass("val3", “abc")
};

Collection<MyClass> list = Arrays.asList(myArray);

list.forEach(c -> System.out.println(c.getLabel()));
```

Collection Types

Java provides a wide variety of collections

- Each with specific behaviors

Collection interfaces

- Provide contract for collection behavior

Collection classes

- Provide collection implementation
- Implement 1 or more collection interfaces

Common Collection Interfaces

Interface	Description
Collection	Basic collection operations
List	Collection that maintains a particular order
Queue	Collection with the concept of order and specific “head” element
Set	Collection that contains no duplicate values
SortedSet	A Set whose members are sorted

Common Collection Classes

Class	Description
ArrayList	A *List* backed by a resizable array Efficient random access but inefficient random inserts
LinkedList	A *List* and *Queue* backed by a doubly-linked list Efficient random insert but inefficient random access
HashSet	A *Set* implemented as a hash table Efficient general purpose usage at any size
TreeSet	A *SortedSet* implemented as a balanced binary tree Members accessible in order but less efficient to modify and search than a HashSet

Sorting

Some collections rely on sorting

- Two ways to specify sort behavior

Comparable interface

- Implemented by the type to be sorted
- Type specifies own sort behavior
  - Should be consistent with equals

Comparator interface

- Implemented by type to perform sort
- Specifies sort behavior for another type

Implementing Comparable

```
public class MyClass implements Comparable<MyClass> {
  String label, value; // Other members elided for clarity
  public String toString() { return label + " | " + value;}

  public boolean equals(Object o) {
    MyClass other = (MyClass) o;
    return value.equalsIgnoreCase(other.value);
  }

  public int compareTo(MyClass other) {
    return value.compareToIgnoreCase(other.value);
  }
}
```

```
- : this < other
0 : this = other
+ : this > other
```

Using TreeSet with Comparable

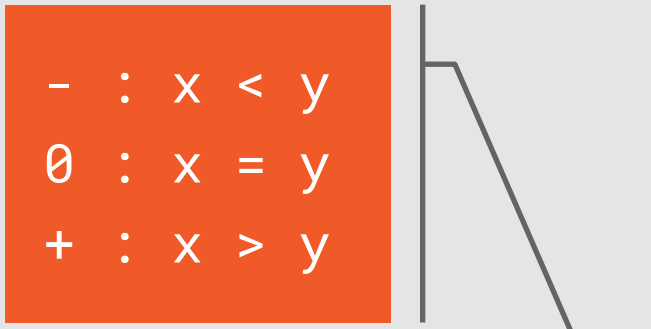
```
TreeSet<MyClass> tree = new TreeSet<>();

tree.add(new MyClass("2222", "ghi"));
tree.add(new MyClass("3333", "abc"));
tree.add(new MyClass("1111", "def"));

tree.forEach(m -> System.out.println(m));
```

```
3333 | abc
1111 | def
2222 | ghi
```

Implementing Comparator




```
public class MyComparator implements Comparator<MyClass> {

  public int compare(MyClass x, MyClass y) {
    return x.getLabel().compareToIgnoreCase(y.getLabel());
  }
}
```

Using TreeSet with Comparator

```
TreeSet<MyClass> tree = new TreeSet<>(new MyComparator());

tree.add(new MyClass("2222", "ghi"));
tree.add(new MyClass("3333", "abc"));
tree.add(new MyClass("1111", "def"));

tree.forEach(m -> System.out.println(m));
```

```
1111 | def
2222 | ghi
3333 | abc
```

Map Collections

Maps store key/value pairs

- Key used to identify/locate values
- Keys are unique
- Values can be duplicated
- Values can be null

Common Map Types

Interface	Description
Map	Basic map operations
SortedMap	Map whose keys are sorted

Class	Description
HashMap	Efficient general purpose *Map* implementation
TreeMap	*SortedMap* implemented as a self-balancing tree Supports *Comparable* and *Comparator* sorting

Common Map Methods

Method	Description
put	Add key and value
putIfAbsent	Add key and value if key not contained or value null
get	Return value for key, if key not found return null
getOrDefault	Return value for key, if key not found return the provided default value
values	Return a *Collection* of the contained values
keySet	Return a *Set* of the contained keys
forEach	Perform action for each entry
replaceAll	Perform action for each entry replacing the each key's value with the action's result

Using Map

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

map.put(“2222", “ghi");
map.put(“3333", “abc");
map.put(“1111", “def");

String s1 = map.get(“3333”);
```

abc

```
String s2 = map.get(“9999”);
```

null

```
String s3 = map.getOrDefault(“9999”, “xyz”);
```

xyz

Using Map

```
Map<String, String> map = new HashMap<>();
map.put(“2222", “ghi");
map.put(“3333", “abc");
map.put(“1111", “def");
map.forEach( (k, v) -> System.out.println(k + “ | ” + v));
```

```
2222 | ghi
3333 | abc
1111 | def
```

```
map.replaceAll( (k, v) -> v.toUpperCase());
map.forEach( (k, v) -> System.out.println(k + “ | ” + v));
```

```
2222 | GHI
3333 | ABC
1111 | DEF
```

Common SortedMap Methods

Method	Description
firstKey	Return first key
lastKey	Return last key
headMap	Return a map for all keys that are less than the specified key
tailMap	Return a map for all keys that are greater than or equal to the specified key
subMap	Return a map for all keys that are greater than or equal to the starting key and less than the ending key

Using SortedMap

```
SortedMap<String, String> map = new TreeMap<>();

map.put(“2222", “ghi");
map.put(“3333", “abc");
map.put(“1111", “def");
map.put(“6666", “xyz");
map.put(“4444", “mno");
map.put(“5555", “pqr");

map.forEach( (k, v) -> System.out.println(k + “ | ” + v));
```

```
1111 | def
2222 | ghi
3333 | abc
4444 | mno
5555 | pqr
6666 | xyz
```

Using SortedMap

```
SortedMap<String, String> map = new TreeMap<>();
// Add same 6 key/value pairs as last slide

SortedMap<String, String> hMap = map.headMap("3333");

hMap.forEach( (k, v) ->
  System.out.println(k + " | " + v));
```

```
1111 | def
2222 | ghi
```

```
SortedMap<String, String> tMap = map.tailMap("3333");

tMap.forEach( (k, v) ->
  System.out.println(k + " | " + v));
```

```
3333 | abc
4444 | mno
5555 | pqr
6666 | xyz
```

Summary

Collections hold and organize values

- Iterable
- Tend to dynamically size
- Can provide optimization or sophistication

Collections can be type restricted

- Uses Java generics to specify type
- Return values appropriately typed
- Typing enforced on added values

Summary

Can convert between collections and arrays

- Collections provide toArray method
- Arrays class' provides toList method

Some collections provide sorting

- Support Comparable interface
  - Type defines own sort
- Support Comparator interface
  - Specifies sort for another type

Summary

Map collections

- Stores key/value pairs
- Keys are unique
- Some maps sort keys