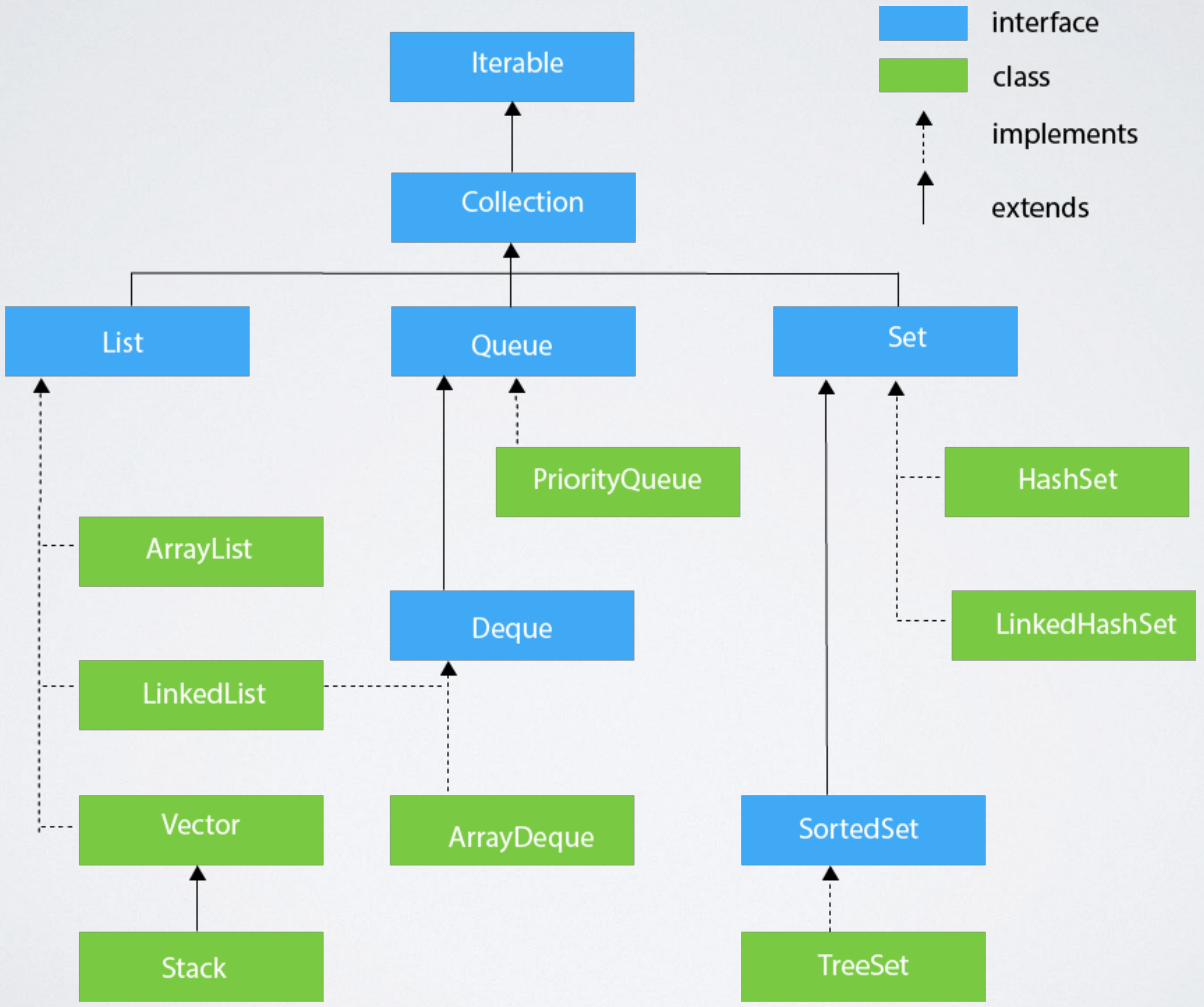
# JAVA BACKEND DEVELOPMENT PROGRAM

Java SE Collection

## OUTLINE

- Collection
- Map
- Ordering
- Java 8

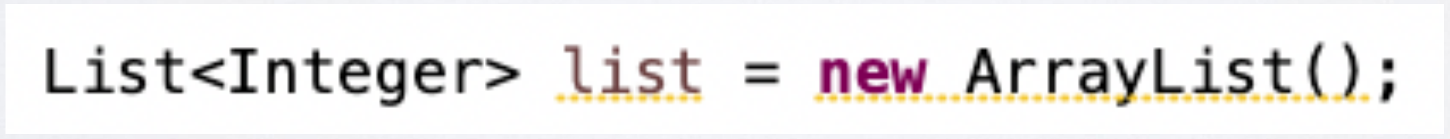
# HIERARCHY



interface
class
implements
extends
Iterable
Collection
List
Queue
Set
PriorityQueue
HashSet
ArrayList
Deque
LinkedHashSet
LinkedList
Vector
ArrayDeque
SortedSet
Stack
TreeSet

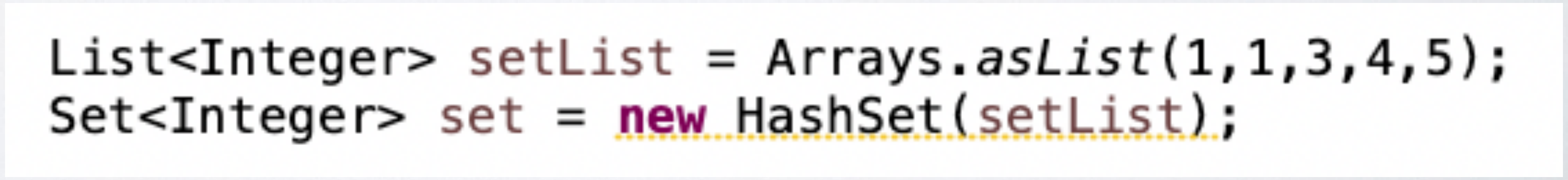
# COLLECTION

- Group of objects
- It is not specified whether they are
  - Ordered / not ordered
  - Duplicated / not duplicated
- Following constructors are common to all classes implementing Collection
  - T()



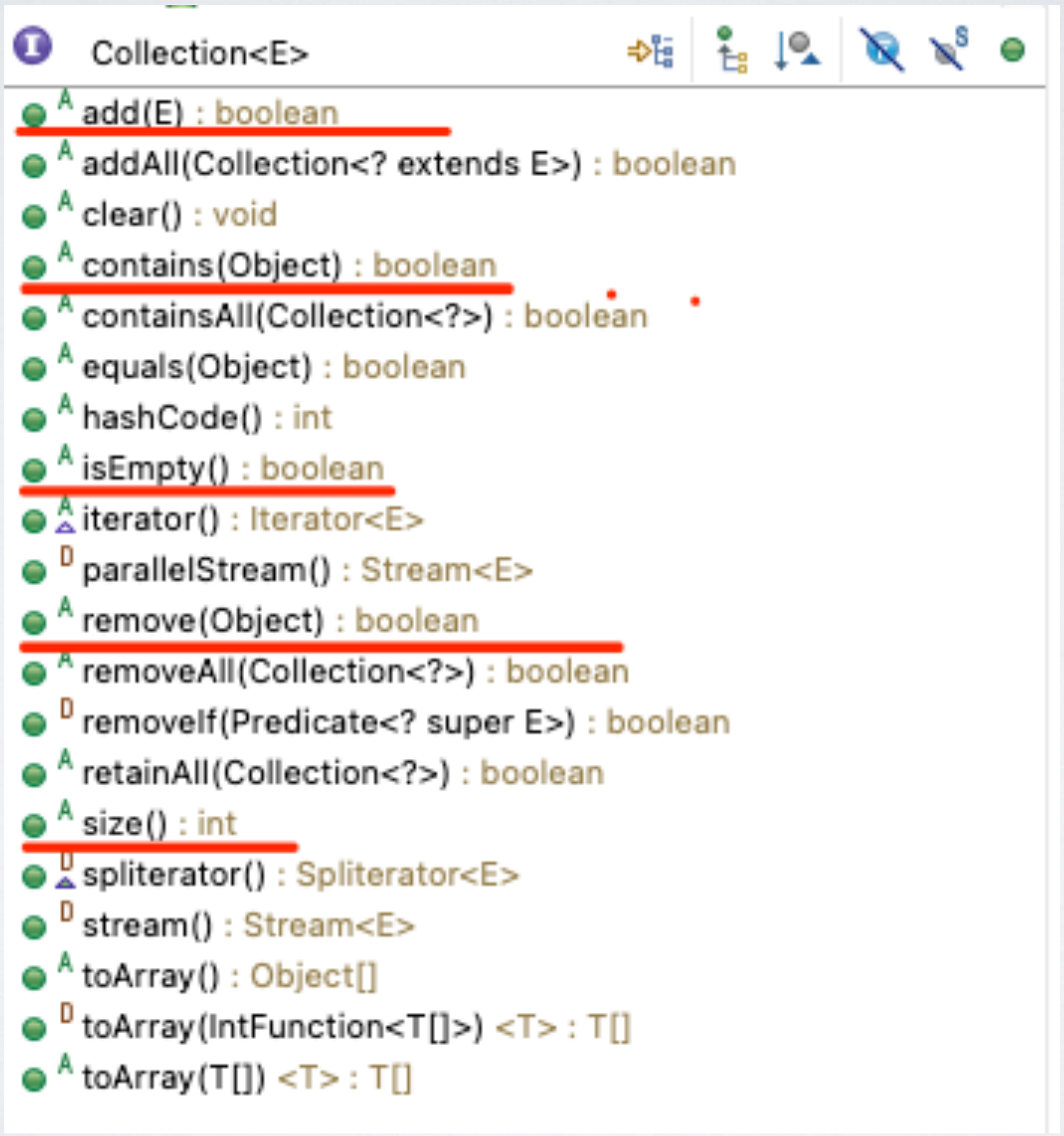
```
List<Integer> list = new ArrayList();
```

  - T(Collection c)



```
List<Integer> setList = Arrays.asList(1,1,3,4,5);
Set<Integer> set = new HashSet(setList);
```

# COLLECTION



```
Collection<E>
add(E) : boolean
addAll(Collection<? extends E>) : boolean
clear() : void
contains(Object) : boolean
containsAll(Collection<?>) : boolean
equals(Object) : boolean
hashCode() : int
isEmpty() : boolean
iterator() : Iterator<E>
parallelStream() : Stream<E>
remove(Object) : boolean
removeAll(Collection<?>) : boolean
removeIf(Predicate<? super E>) : boolean
retainAll(Collection<?>) : boolean
size() : int
spliterator() : Spliterator<E>
stream() : Stream<E>
toArray() : Object[]
toArray(IntFunction<T[]>) <T> : T[]
toArray(T[]) <T> : T[]
```

# LIST

- Can contain duplicate elements
- Homogeneous
- Insertion order is preserved
- User can define insertion point
- Elements can be accessed by position

# LIST ADDITIONAL METHOD

- `Object get(int index)`
- `Object set(int index, Object element)`
- `void   add(int index, Object element)`
- `Object remove(int index)`

- `boolean addAll(int index, Collection c)`
- `int indexOf(Object o)`
- `int lastIndexOf(Object o)`
- `List subList(int fromIndex, int toIndex)`

## LIST IMPLEMENTATION

ArrayList

- get(n)
  - Constant time
- Insert (beginning) and delete while iterating
  - Linear

**[1,2,3,4,5]**
**[1,3,4,5]**

LinkedList

- get(n)
  - Linear time
- Insert (beginning) and delete while iterating
  - Constant

**1->2->3->4->5**
**1->3->4->5**

## QUEUE

- Collection whose elements have an order
  - FIFO: First In First Out
- Defines a head position where is the first element that can be accessed
  - offer()
  - peek()
  - poll()

## QUEUE IMPLEMENTATION

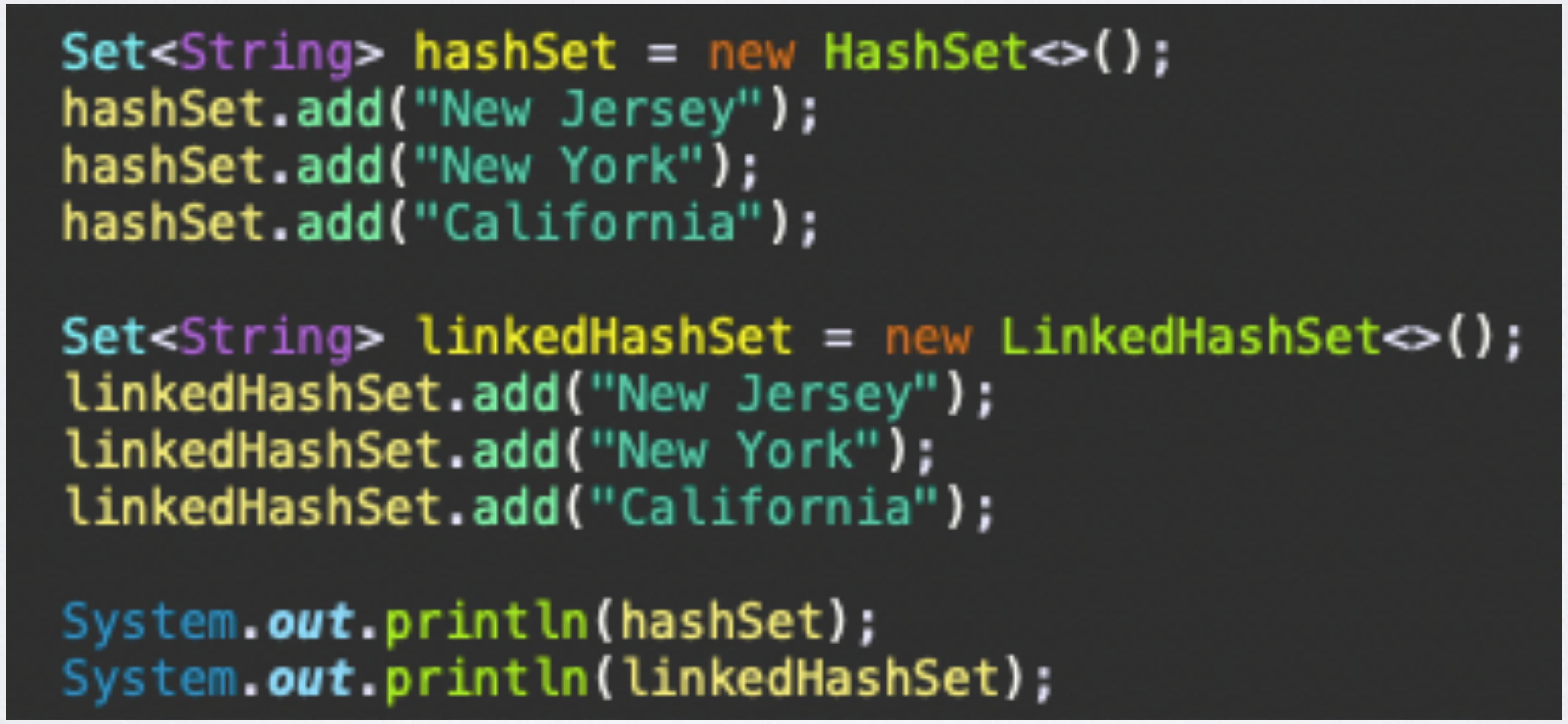
- LinkedList
- PriorityQueue

## SET

- Contains no methods other than those inherited from Collection
- add() has restriction that no duplicate elements are allowed

# SET IMPLEMENTATION

- HashSet
  - Insertion order is **not** preserved
- LinkedHashSet
  - Insertion order is preserved

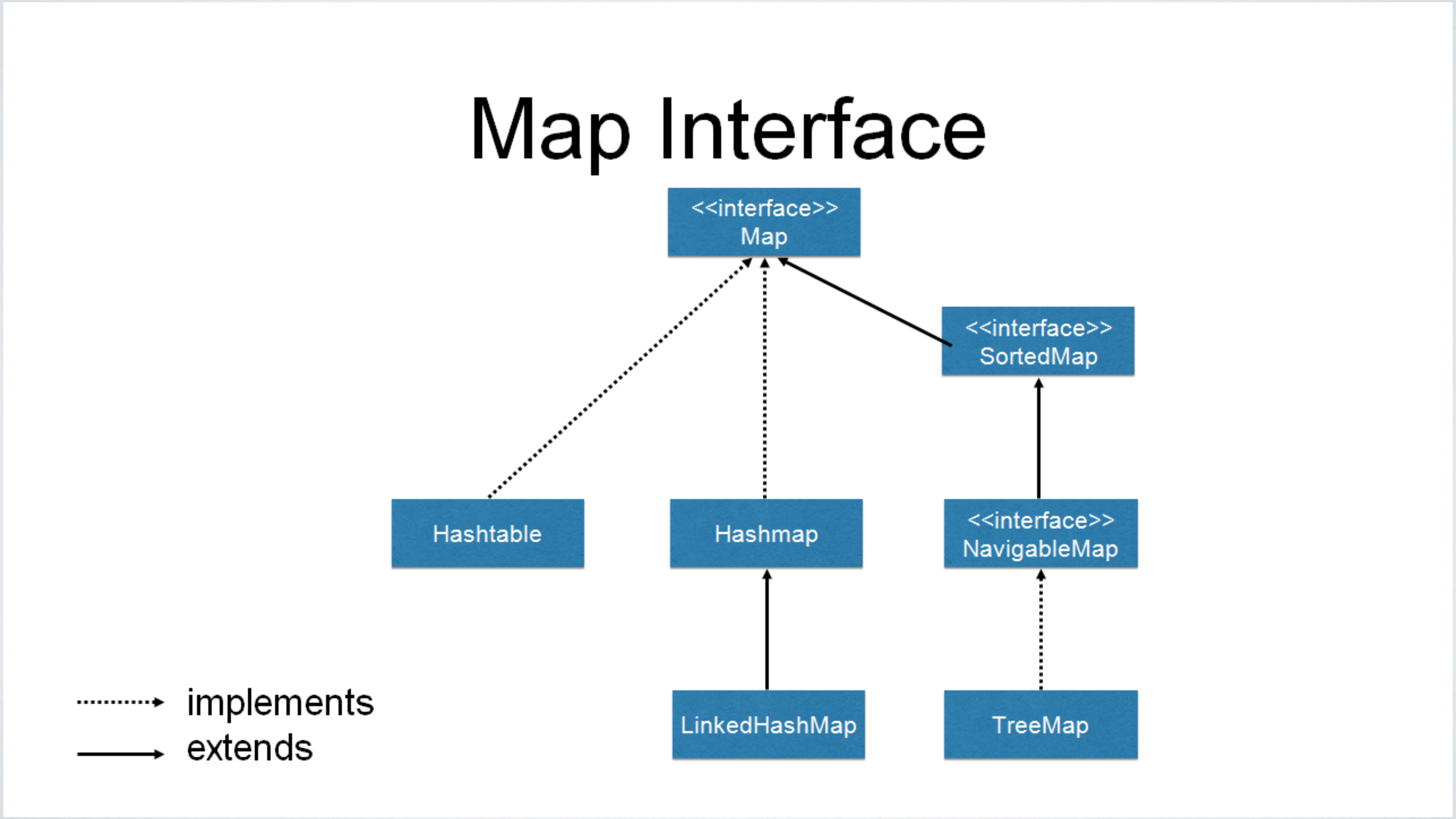


```java
Set<String> hashSet = new HashSet<>();
hashSet.add("New Jersey");
hashSet.add("New York");
hashSet.add("California");

Set<String> linkedHashSet = new LinkedHashSet<>();
linkedHashSet.add("New Jersey");
linkedHashSet.add("New York");
linkedHashSet.add("California");

System.out.println(hashSet);
System.out.println(linkedHashSet);
```

## OUTLINE

- Collection
- Map
- Ordering
- Java 8

# HIERARCHY


Map Interface
<<interface>> Map
<<interface>> SortedMap
Hashtable
Hashmap
<<interface>> NavigableMap
LinkedHashMap
TreeMap
implements
extends

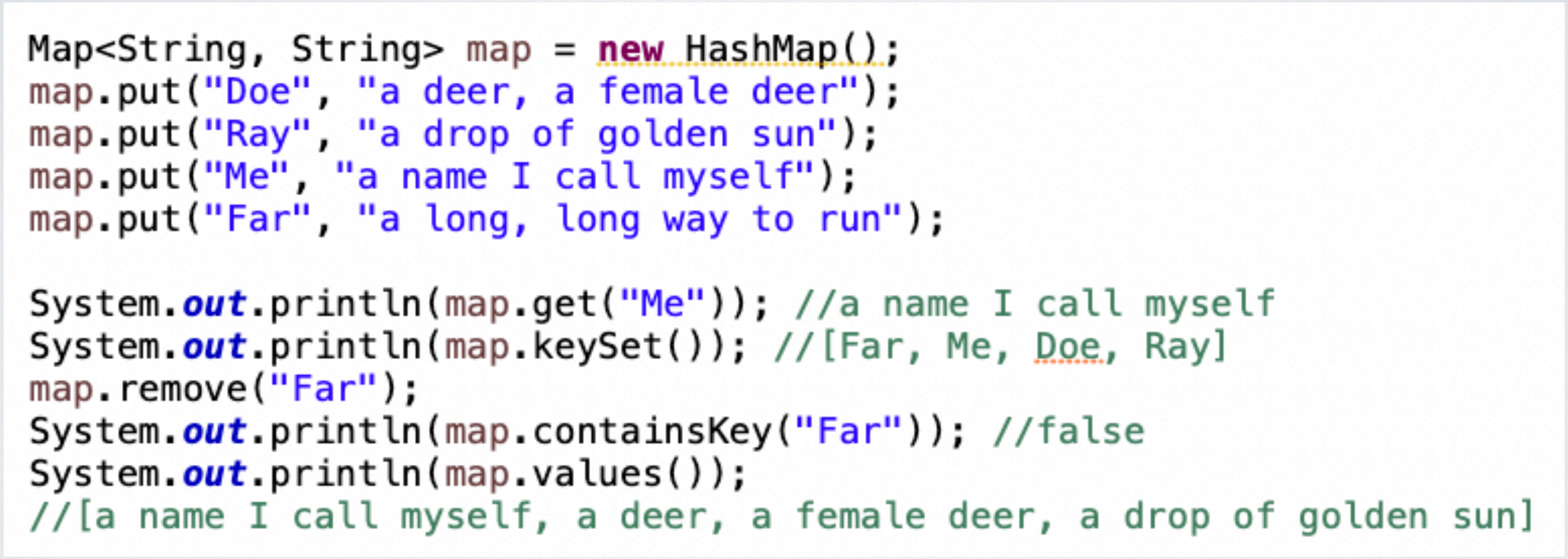
## MAP

- An object that associates keys to values
- Keys and values must be objects(Wrapper Class)
- Keys must be unique
- Only one value per key

## MAP INTERFACE

- **`Object put(Object key, Object value)`**
- **`Object get(Object key)`**
- **`Object remove(Object key)`**
- **`boolean containsKey(Object key)`**
- **`boolean containsValue(Object value)`**
- **`public Set keySet()`**
- **`public Collection values()`**
- **`int size()`**
- **`boolean isEmpty()`**
- **`void clear()`**

# MAP



```java
Map<String, String> map = new HashMap();
map.put("Doe", "a deer, a female deer");
map.put("Ray", "a drop of golden sun");
map.put("Me", "a name I call myself");
map.put("Far", "a long, long way to run");

System.out.println(map.get("Me")); //a name I call myself
System.out.println(map.keySet()); //[Far, Me, Doe, Ray]
map.remove("Far");
System.out.println(map.containsKey("Far")); //false
System.out.println(map.values());
//[a name I call myself, a deer, a female deer, a drop of golden sun]
```

## EQUALS(). ==

- ==
  - Address comparison
  - If point to the same memory location
- Equals:
  - Values in the object

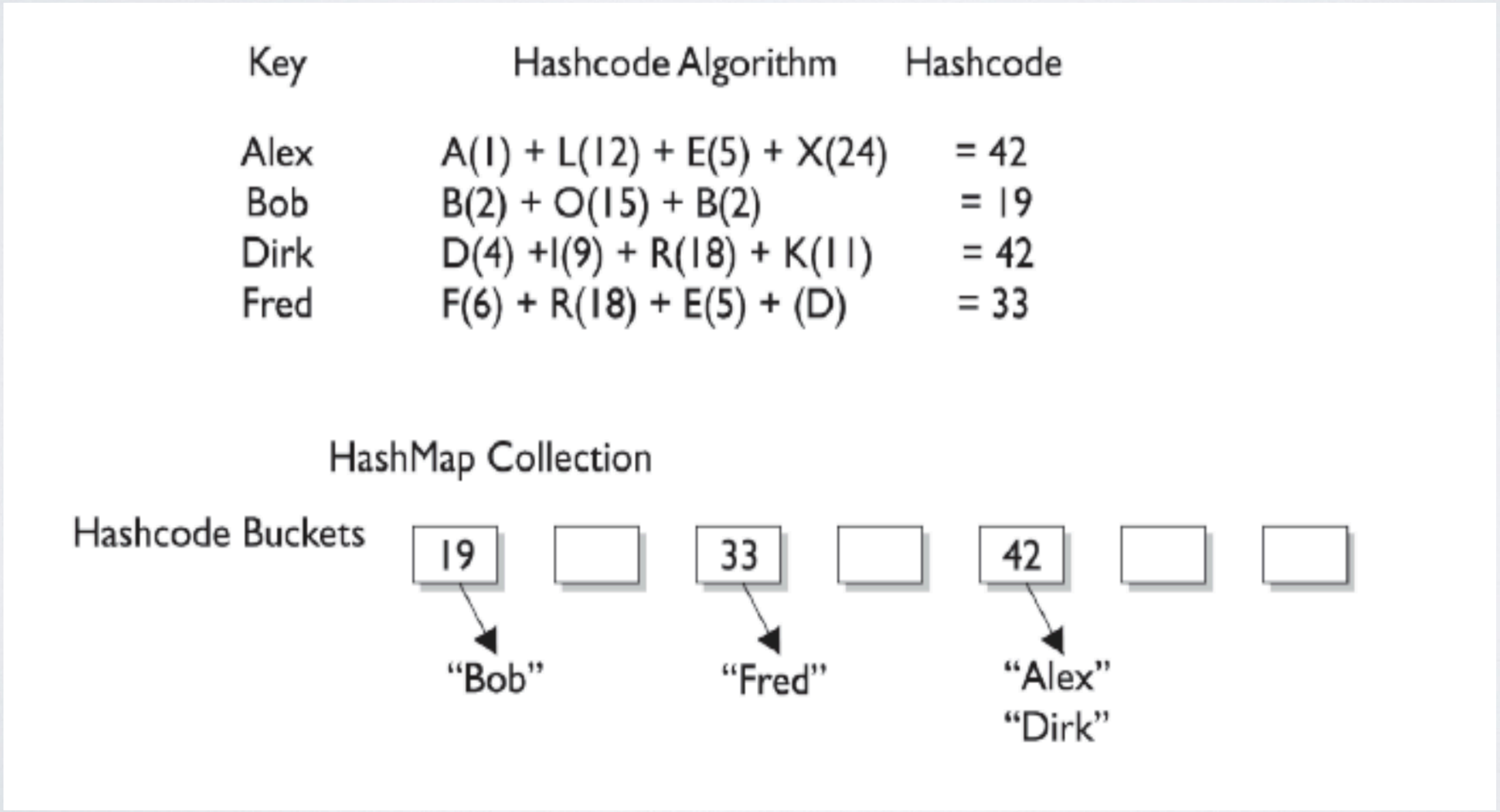
# EQUALS()

- It is reflexive: x.equals(x) == true
- It is symmetric: x.equals(y) == y.equals(x)
- It is transitive: for any reference values x, y, and z, if x.equals(y) == true AND y.equals(z) == true => x.equals(z) == true
- It is consistent: for any reference values x and y, multiple invocations of x.equals(y) consistently return true (or false), provided that no information used in equals comparisons on the object is modified.
- x.equals(null) == false

# HASHCODE()

- Return Type: int

- Object.hashCode()



| Key | Hashcode Algorithm | Hashcode |
|---|---|---|
| Alex | A(1) + L(12) + E(5) + X(24) | = 42 |
| Bob | B(2) + O(15) + B(2) | = 19 |
| Dirk | D(4) +I(9) + R(18) + K(11) | = 42 |
| Fred | F(6) + R(18) + E(5) + (D) | = 33 |

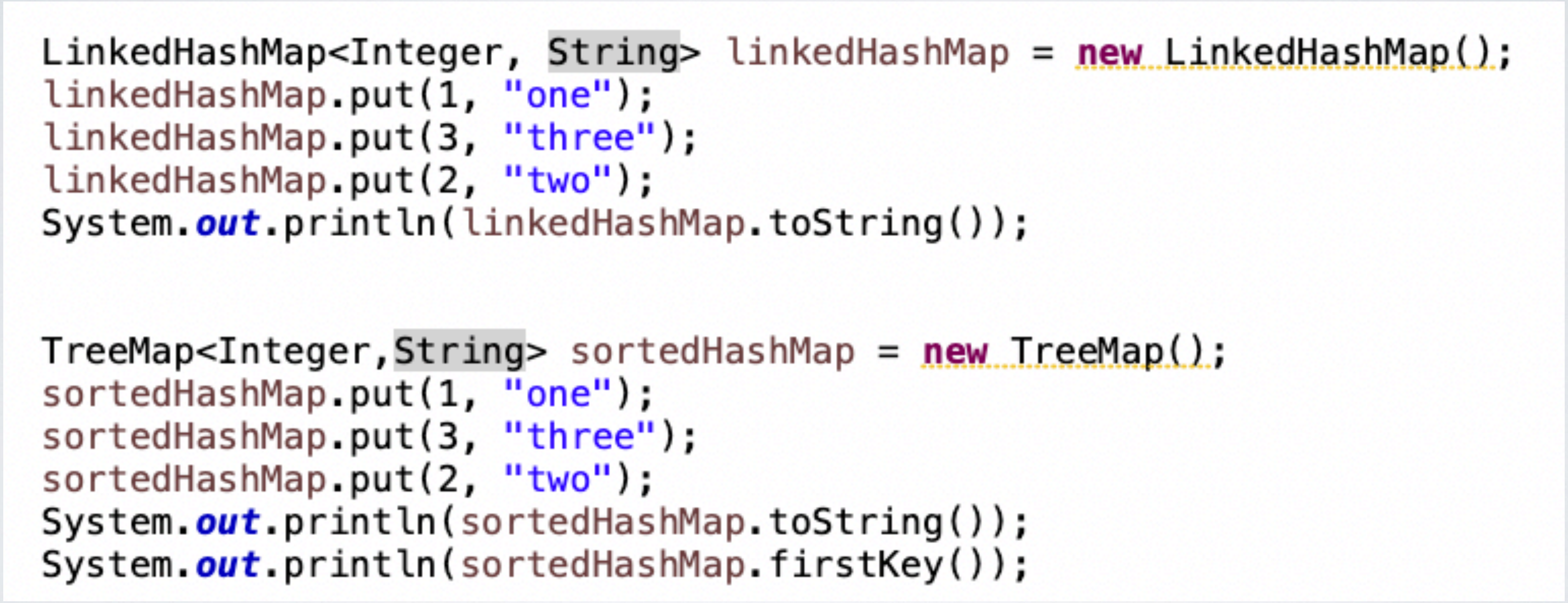
# HASHCODE()

- The hashCode() method must consistently return the same int, if information used in equals() to compare the objects is not modified.
- The hashCode contract in Java 8:
  - If two objects are equal for equals() method, then calling the hashCode() method on the two objects must produce the same integer result.
  - If two objects are unequal for equals() method, then calling the hashCode() method on the two objects MAY produce distinct integer results.
  - Producing distinct int results for unequal objects may improve the performance of hashmap

## MAP

- Get/set takes constant time (in case of no collisions)
- Implementations
  - HashMap implements Map
  - LinkedHashMap extends HashMap
  - TreeMap implements SortedMap

# SORTED MAP



```java
LinkedHashMap<Integer, String> linkedHashMap = new LinkedHashMap();
linkedHashMap.put(1, "one");
linkedHashMap.put(3, "three");
linkedHashMap.put(2, "two");
System.out.println(linkedHashMap.toString());


TreeMap<Integer,String> sortedHashMap = new TreeMap();
sortedHashMap.put(1, "one");
sortedHashMap.put(3, "three");
sortedHashMap.put(2, "two");
System.out.println(sortedHashMap.toString());
System.out.println(sortedHashMap.firstKey());
```

# OUTLINE

- Collection
- Map
- Ordering
- Java 8

# OBJECT ODERRING

- How to sort collections
  - Bubble sort
  - Selection sort
  - Quick sort
  - Merge sort
- Take the advantage of Collection

# COLLECTIONS

- Static methods of java.util.Collections class
  - sort() - Tim sort, n log(n)
  - binarySearch() – requires ordered sequence
  - reverse() - requires ordered sequence
  - min(), max() – in a Collection

## DEFAULT IMPLEMENTATION

- The interface is implemented by language common types in packages java.lang and java.util
  - String objects are lexicographically ordered
  - Date objects are chronologically ordered
  - Number and sub-classes are ordered numerically

## CUSTOM ORDERING

- Comparable<T> interface
- Comparator<T> interface

# COMPARABLE INTERFACE

```
public interface Comparable<T> {
    public int compareTo(T obj);
}
```

- Compares the receiving object with the specified object
- Return value must be:
  - <0, if this precedes obj
  - ==0, if this has the same order as obj
  - >0, if this follows obj

# COMPARATOR INTERFACE

- Compares its two arguments
- Return value must be
  - <0, if o1 precedes o2
  - ==0, if o1 has the same ordering as o2
  - >0, if o1 follows o2

## OUTLINE

- Collection
- Map
- Iterator
- Ordering
- Java 8

## JAVA 8 NEW FEATURES

- Lambda Expression
- Functional Interface

## LAMBDA EXPRESSION

- Simplifies development
- Before Java 8, there is anonymous class
- Syntax
  - parameter -> expression

## FUNCTIONAL INTERFACE

- Interface that contains ONLY ONE **abstract** method
  - Comparator is a functional interface

## LAMBDA EXPRESSION + FUNCTIONAL INTERFACE

- Lambda Expression and Functional Interface works together to simplify code

## QUESTIONS?