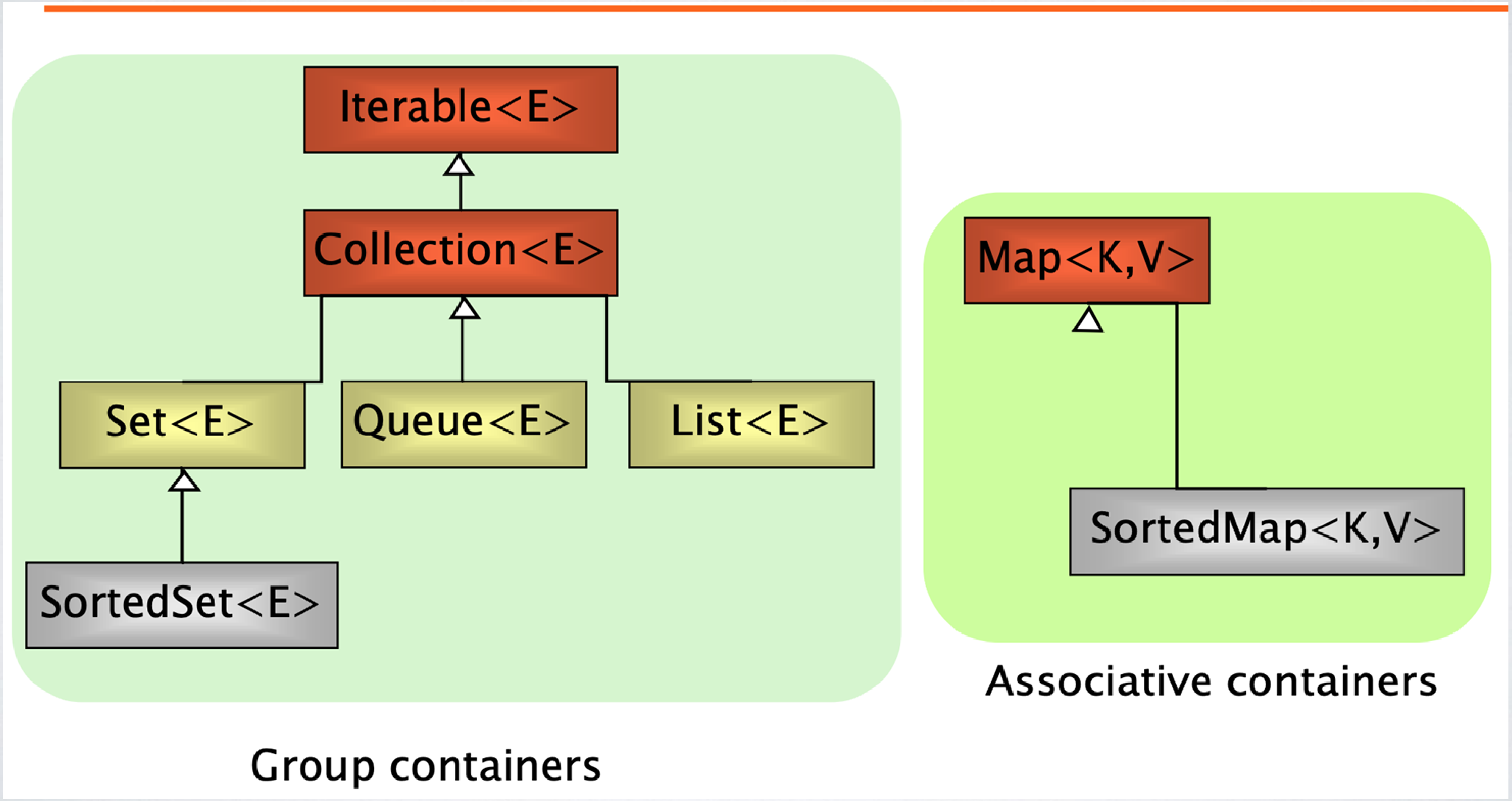
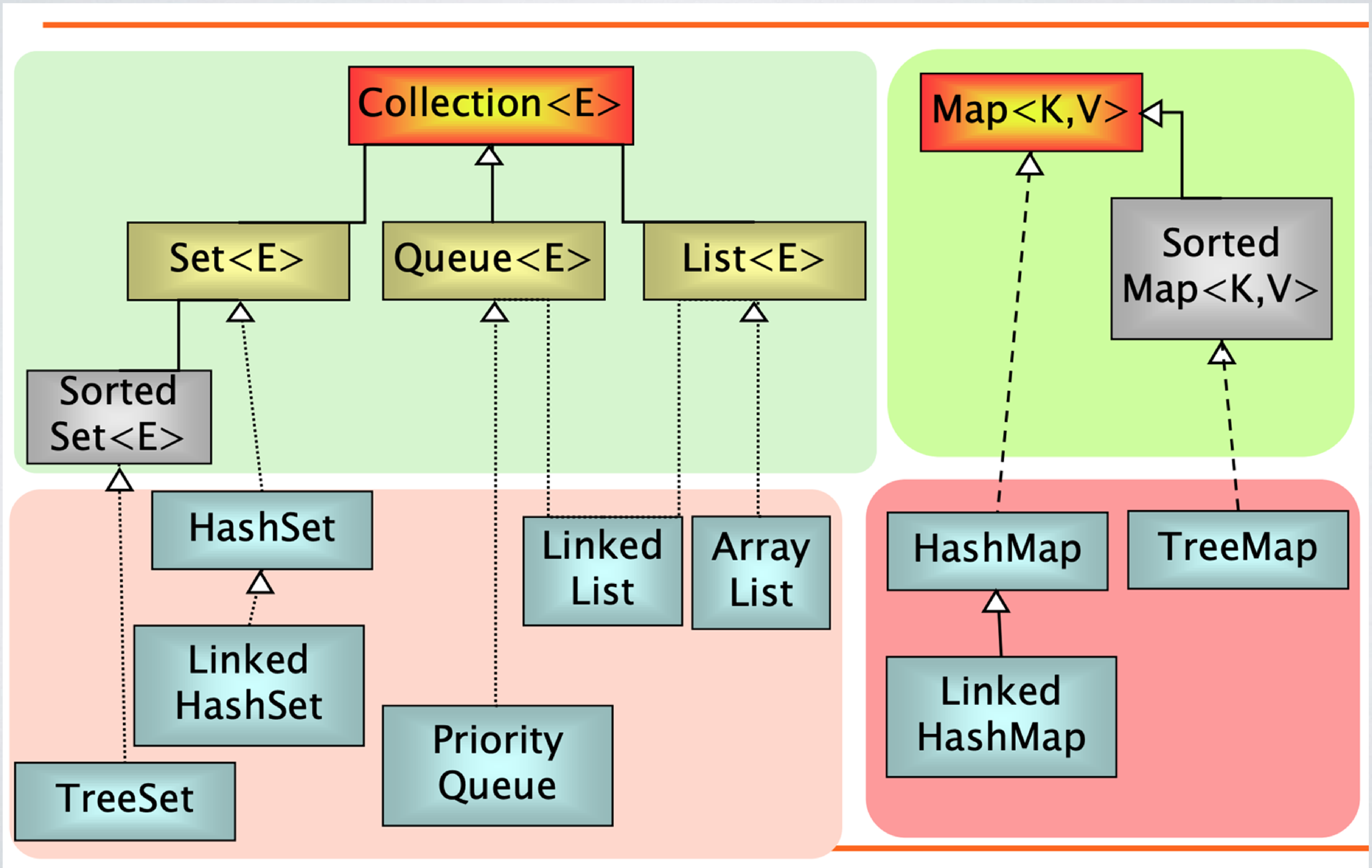
JAVA FULL STACK DEVELOPMENT PROGRAM

Session 5: Java SE Collection

INTERFACE

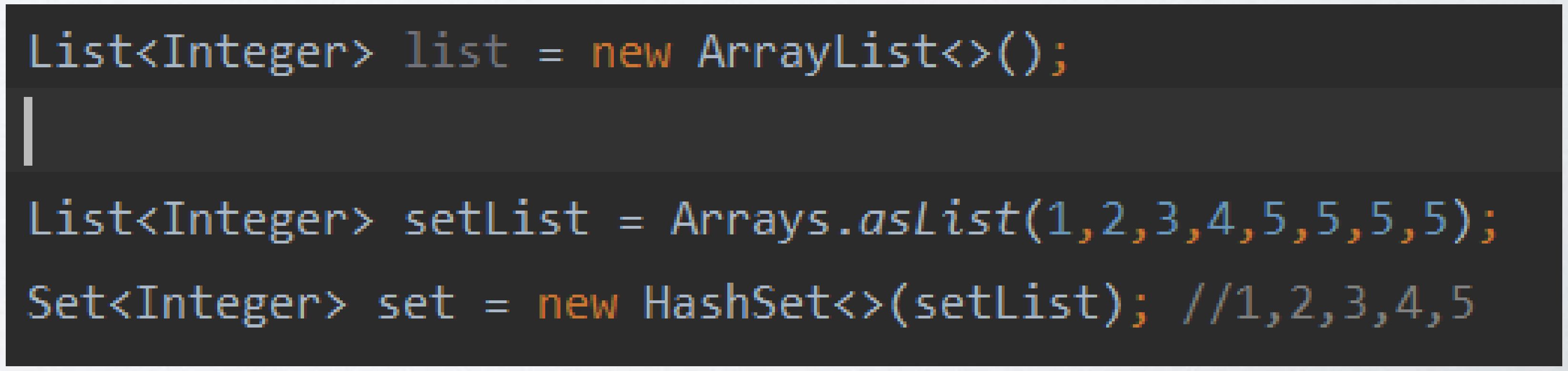

Iterable<E>
Collection<E>
Set<E>
Queue<E>
List<E>
SortedSet<E>
Group containers
Map<K,V>
SortedMap<K,V>
Associative containers

IMPLEMENTATIONS



COLLECTION

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



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

List<Integer> setList = Arrays.asList(1,2,3,4,5,5,5,5);
Set<Integer> set = new HashSet<>(setList); //1,2,3,4,5
```

COLLECTION INTERFACE

- `int size()`
- `boolean isEmpty()`
- `boolean contains(Object element)`
- `boolean containsAll(Collection c)`
- `boolean add(Object element)`
- `boolean addAll(Collection c)`
- `boolean remove(Object element)`
- `boolean removeAll(Collection c)`
- `void clear()`
- `Object[] toArray()`
- `Iterator iterator()`

LIST

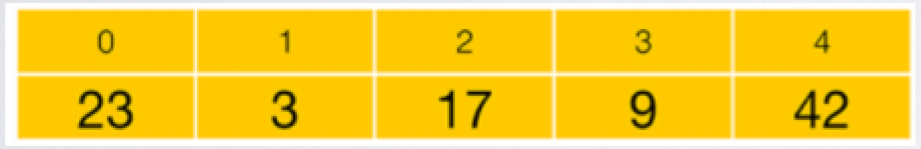
- Can contain duplicate elements
- Insertion order is preserved
- User can define insertion point
- Elements can be accessed by position

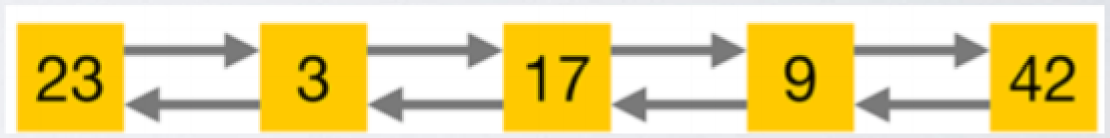
LIST ADDITIONAL METHOD

- `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	LinkedList
get(index): O(1)	get(index): O(N)
add(index,element) if index = 0: O(n)	addFirst(), getFirst() : O(1)
add(element): O(1)	addLast() getLast(): O(1)
remove(index): O(n)	remove(index): O(n)
	remove(Node): O(1)

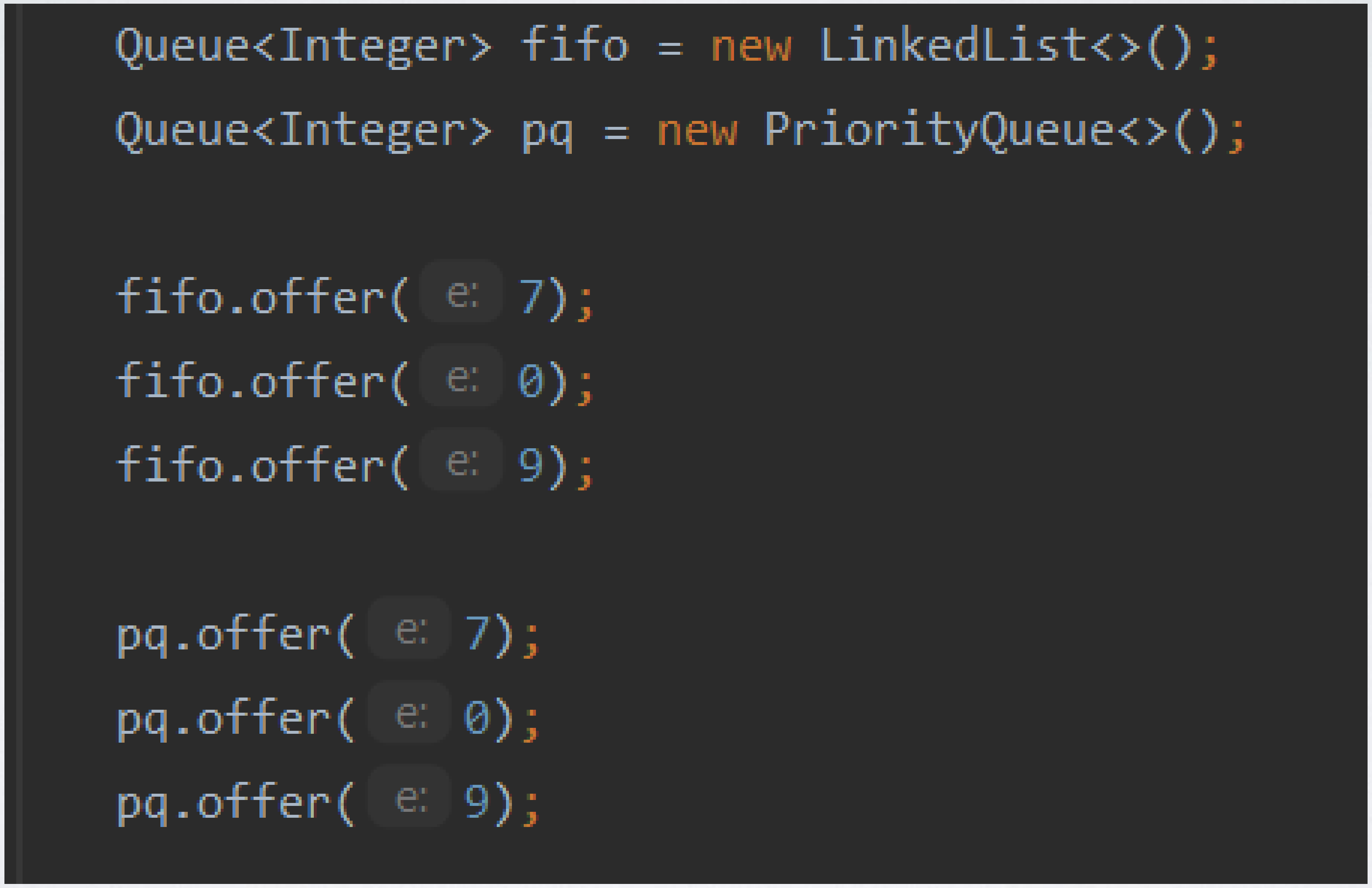
QUEUE

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

QUEUE IMPLEMENTATION

- LinkedList
    - Head is the first element of the list
- PriorityQueue
    - Head is the smallest/largest element

QUEUE IMPLEMENTATION



```
Queue<Integer> fifo = new LinkedList<>();
Queue<Integer> pq = new PriorityQueue<>();

fifo.offer( e: 7);
fifo.offer( e: 0);
fifo.offer( e: 9);

pq.offer( e: 7);
pq.offer( e: 0);
pq.offer( e: 9);
```

SET

- Contains no methods other than those inherited from Collection
- add()has restriction that no duplicate elements are allowed
- Iterator
  - The elements are traversed in no particular order

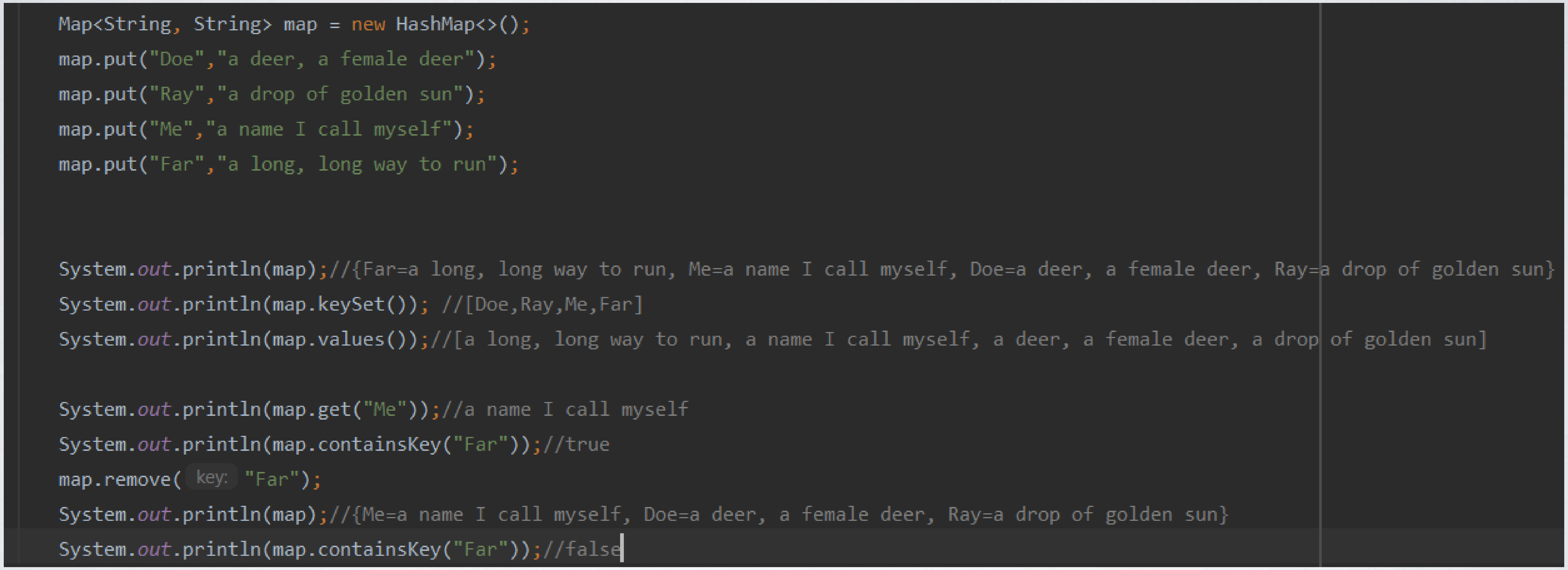
MAP

- An object that associates keys to values
- Keys and values must be objects
- Keys must be unique (how about null?)
- Only one value per key
- Following constructors are common to all collection implementers
    - T()
    - T(Map map)

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



```
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);//{Far=a long, long way to run, Me=a name I call myself, Doe=a deer, a female deer, Ray=a drop of golden sun}
System.out.println(map.keySet()); //[Doe,Ray,Me,Far]
System.out.println(map.values());//[a long, long way to run, a name I call myself, a deer, a female deer, a drop of golden sun]

System.out.println(map.get("Me"));//a name I call myself
System.out.println(map.containsKey("Far"));//true
map.remove( key: "Far");
System.out.println(map);//{Me=a name I call myself, Doe=a deer, a female deer, Ray=a drop of golden sun}
System.out.println(map.containsKey("Far"));//false
```

equals() vs ==

- ==:
  - Reference comparison (Address Comparison)
  - If point to the same memory location

- equals():
  - Default implementation: simply checks if two Object references (say x and y) refer to the same Object. if it is a String, it will check if the two strings are same.

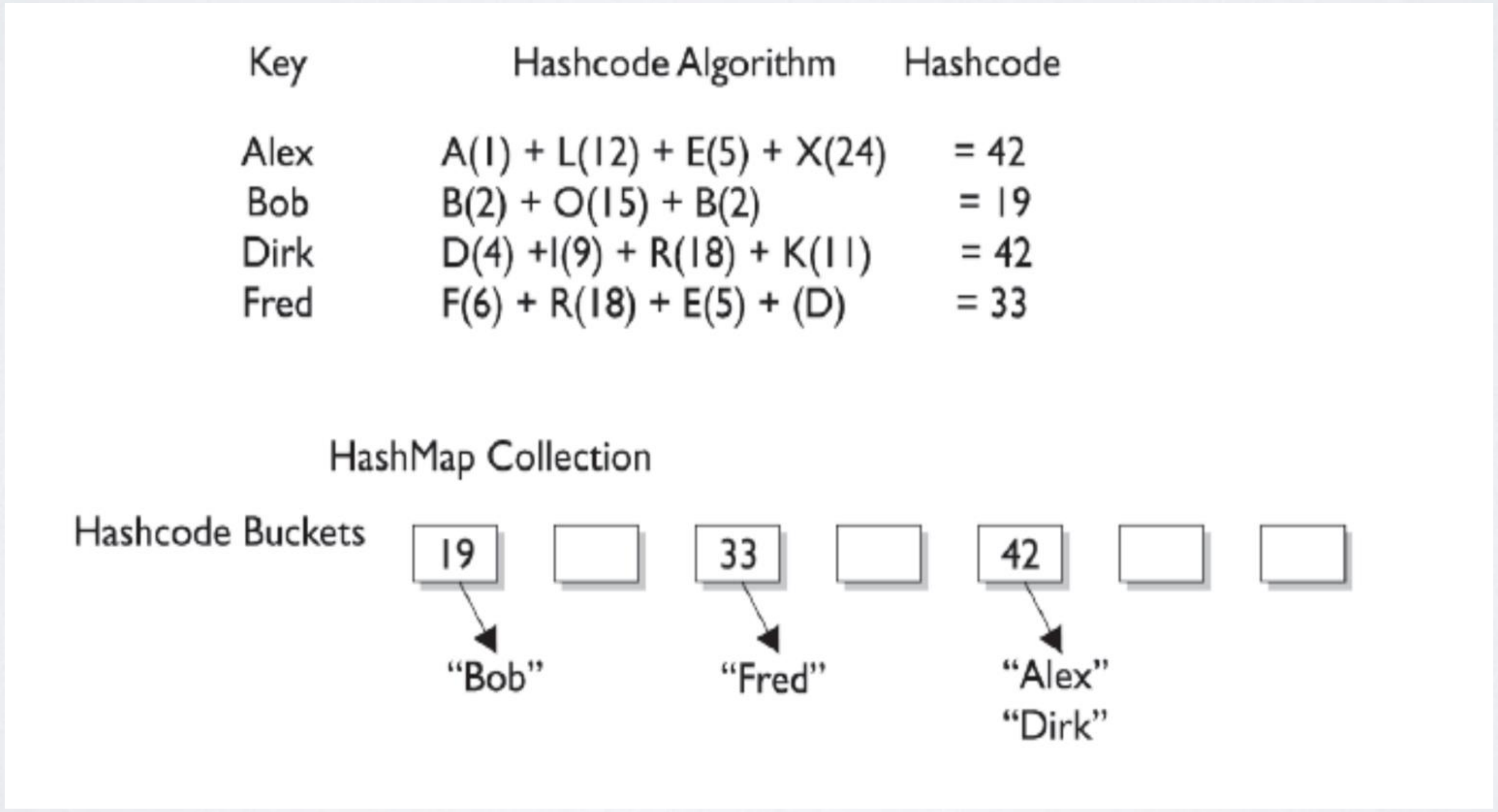
  - We could use @Override to compare the values in the object

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 no information used in equals() comparisons on the object is modified.
- 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 hashtables
    - The default implementation provided by the JDK is based on memory location — two objects are equal if and only if they are stored in the same memory address.

MAP

- Analogous of Set
- Get/set takes constant time (On average)
- Implementations
    - HashMap implements Map
    - LinkedHashMap extends HashMap
    - TreeMap implements SortedMap

HOW IS HASHMAP IMPLEMENTED IN JAVA

- Hashing:

  Hashing is a process of converting an object into integer form by using the method hashCode().

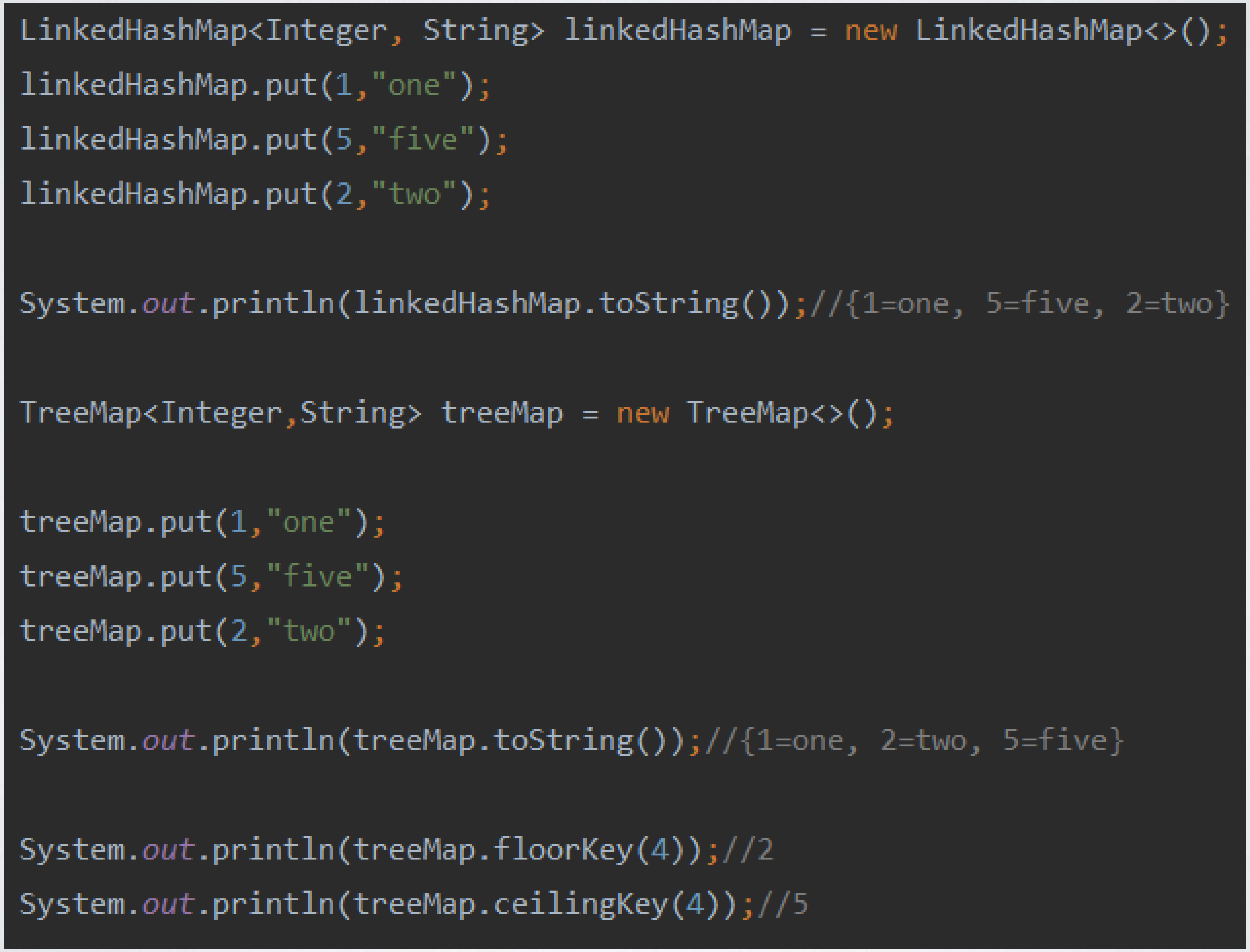
- Buckets:

  A bucket is one element of HashMap array. It is used to store nodes. Two or more nodes can have the same bucket. In that case link list structure is used to connect the nodes.

SORTED MAP

- The elements are traversed according to the keys’ ordering
- TreeMap
- LinkedHashMap is not a sorted map, but it will keep the order of insertion.

SORTED MAP



```
LinkedHashMap<Integer, String> linkedHashMap = new LinkedHashMap<>();
linkedHashMap.put(1,"one");
linkedHashMap.put(5,"five");
linkedHashMap.put(2,"two");

System.out.println(linkedHashMap.toString());//{1=one, 5=five, 2=two}

TreeMap<Integer,String> treeMap = new TreeMap<>();

treeMap.put(1,"one");
treeMap.put(5,"five");
treeMap.put(2,"two");

System.out.println(treeMap.toString());//{1=one, 2=two, 5=five}

System.out.println(treeMap.floorKey(4));//2
System.out.println(treeMap.ceilingKey(4));//5
```

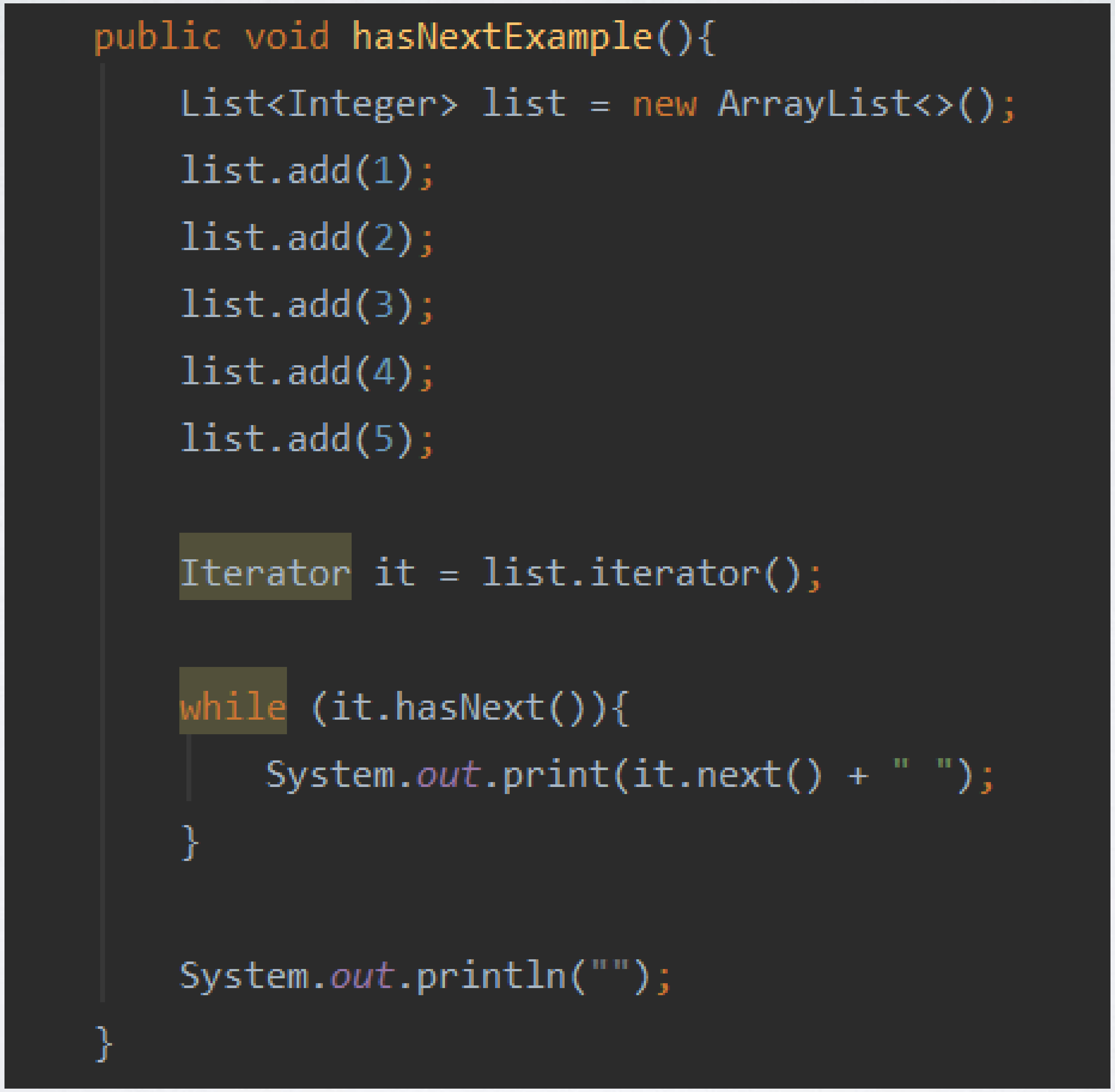
ITERATOR

- A common operation with collections is to iterate over their elements
- Interface Iterator provides a transparent means to cycle through all elements of a Collection
- Keeps track of last visited element of the related collection
- Each time the current element is queried, it moves on automatically

ITERATOR INTERFACE

- boolean hasNext()
- Object next()
- void remove()

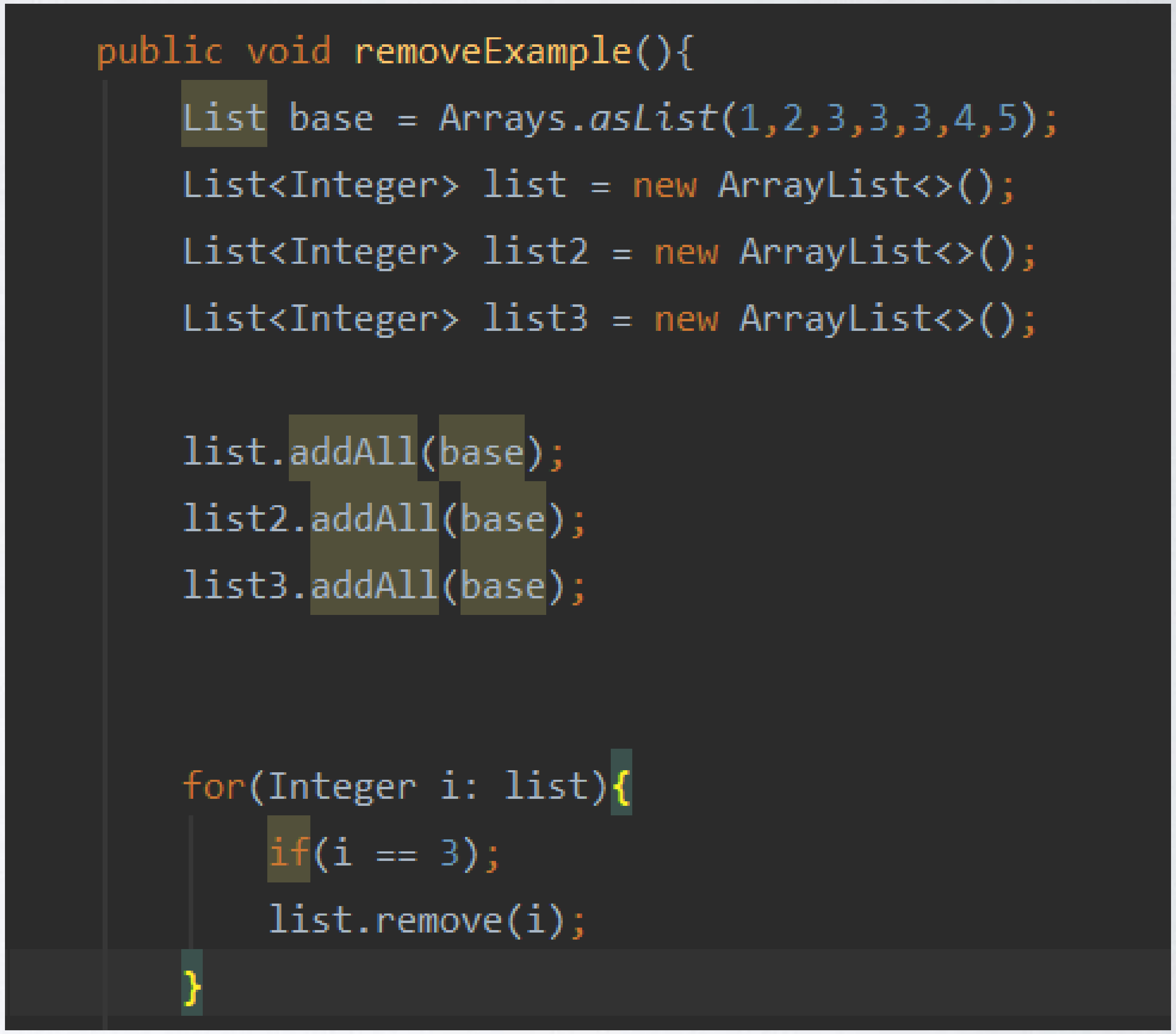
ITERATOR



```java
public void hasNextExample(){
    List<Integer> list = new ArrayList<>();
    list.add(1);
    list.add(2);
    list.add(3);
    list.add(4);
    list.add(5);

    Iterator it = list.iterator();

    while (it.hasNext()){
        System.out.print(it.next() + " ");
    }

    System.out.println("");
}
```

ITERATOR

- Why iterator?
- Considering following code

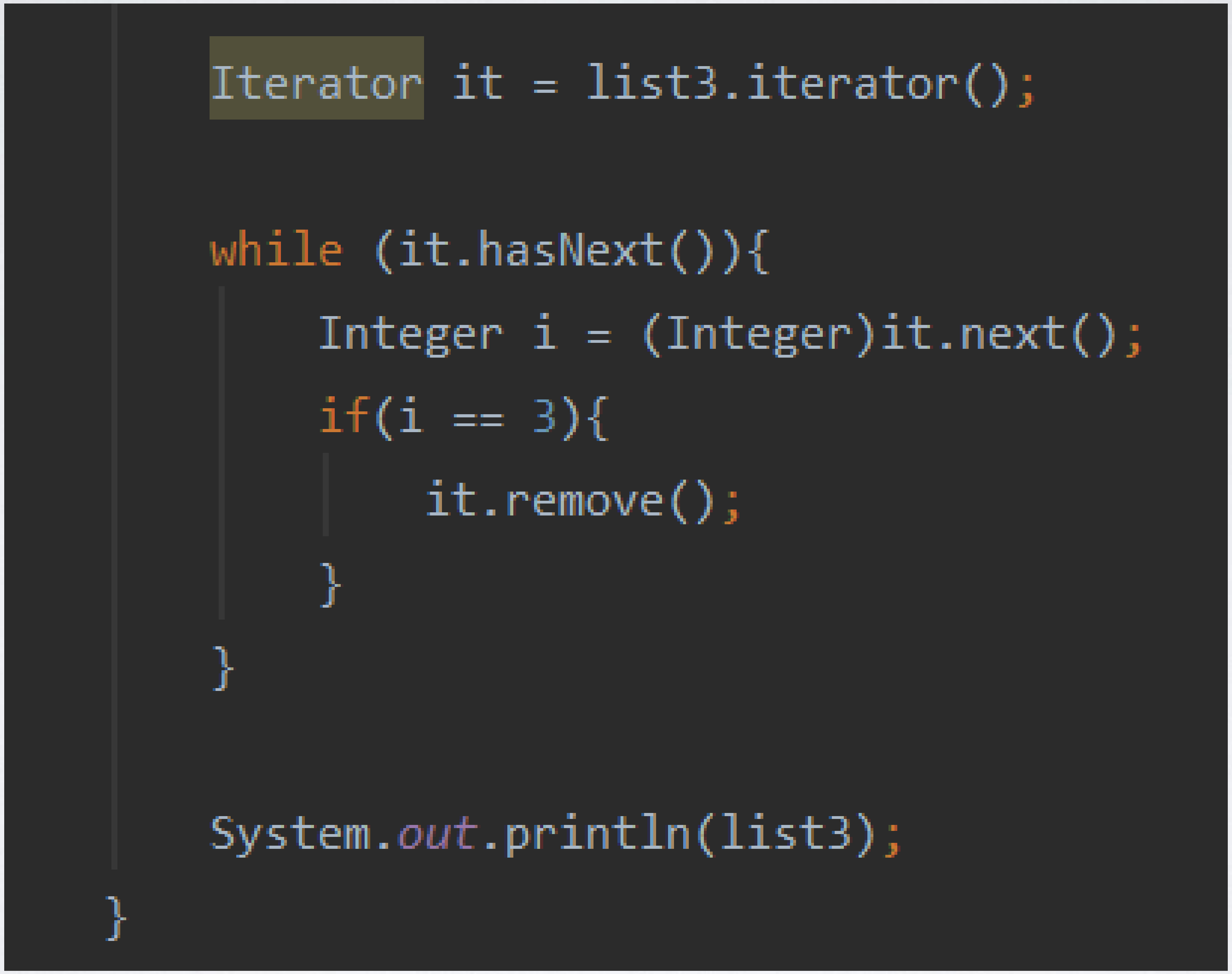


```java
public void removeExample(){
    List base = Arrays.asList(1,2,3,3,3,4,5);
    List<Integer> list = new ArrayList<>();
    List<Integer> list2 = new ArrayList<>();
    List<Integer> list3 = new ArrayList<>();

    list.addAll(base);
    list2.addAll(base);
    list3.addAll(base);


    for(Integer i: list){
        if(i == 3);
        list.remove(i);
    }
```

ITERATOR

- It is unsafe to iterate over a collection you are modifying (add/del) at the same time
- Unless you are using the iterator methods
    - Iterator.remove()
    - ListIterator.add()

ITERATOR



```
    Iterator it = list3.iterator();

    while (it.hasNext()){
        Integer i = (Integer)it.next();
        if(i == 3){
            it.remove();
        }
    }

    System.out.println(list3);
}
```

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

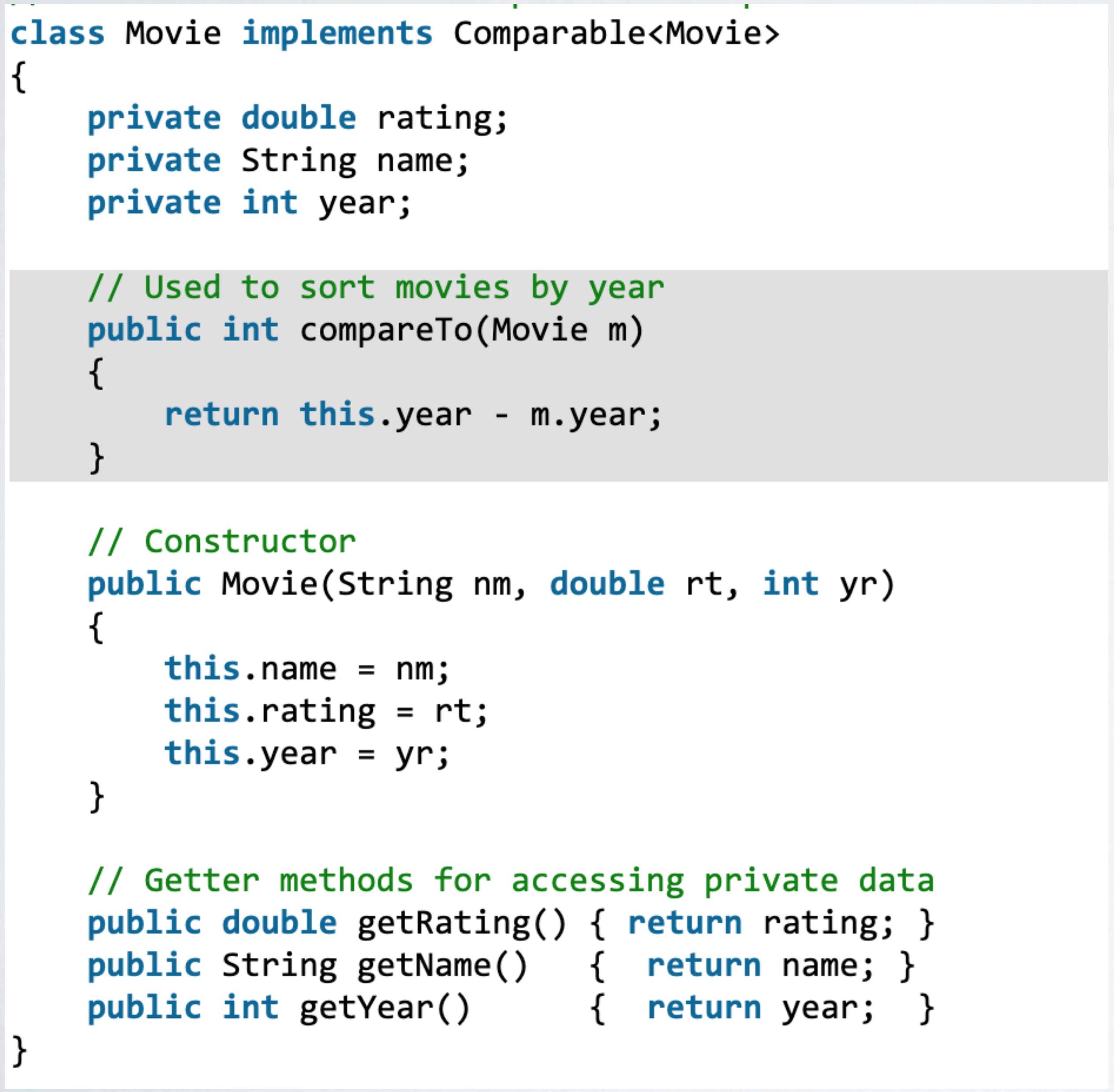
COMPARATOR INTERFACE

- 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

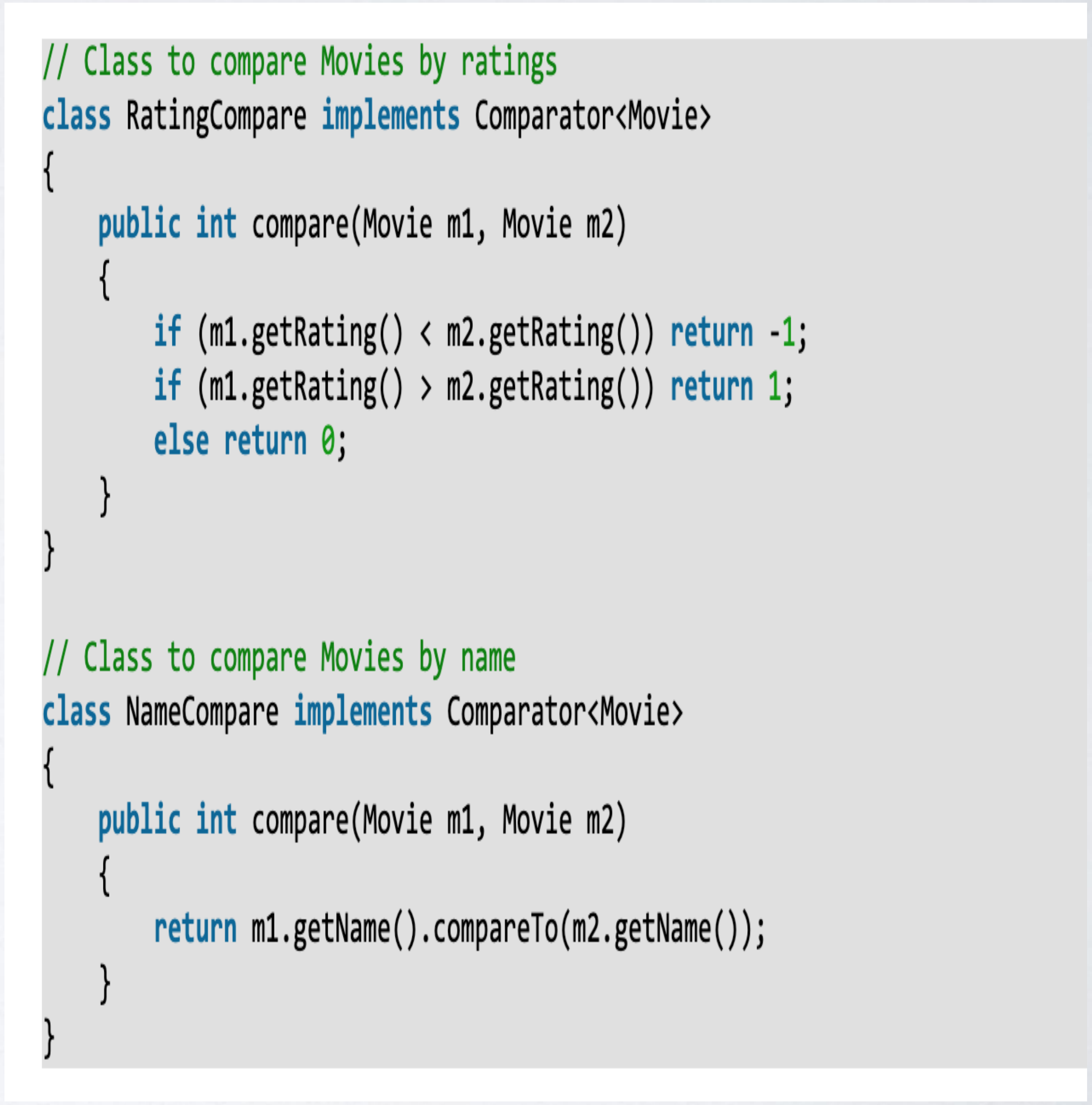
DEFAULT IMPLEMENTATION

- 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



```java
class Movie implements Comparable<Movie>
{
    private double rating;
    private String name;
    private int year;

    // Used to sort movies by year
    public int compareTo(Movie m)
    {
        return this.year - m.year;
    }

    // Constructor
    public Movie(String nm, double rt, int yr)
    {
        this.name = nm;
        this.rating = rt;
        this.year = yr;
    }

    // Getter methods for accessing private data
    public double getRating() { return rating; }
    public String getName()   {  return name; }
    public int getYear()      {  return year;  }
}
```



```java
// Class to compare Movies by ratings
class RatingCompare implements Comparator<Movie>
{
    public int compare(Movie m1, Movie m2)
    {
        if (m1.getRating() < m2.getRating()) return -1;
        if (m1.getRating() > m2.getRating()) return 1;
        else return 0;
    }
}

// Class to compare Movies by name
class NameCompare implements Comparator<Movie>
{
    public int compare(Movie m1, Movie m2)
    {
        return m1.getName().compareTo(m2.getName());
    }
}
```

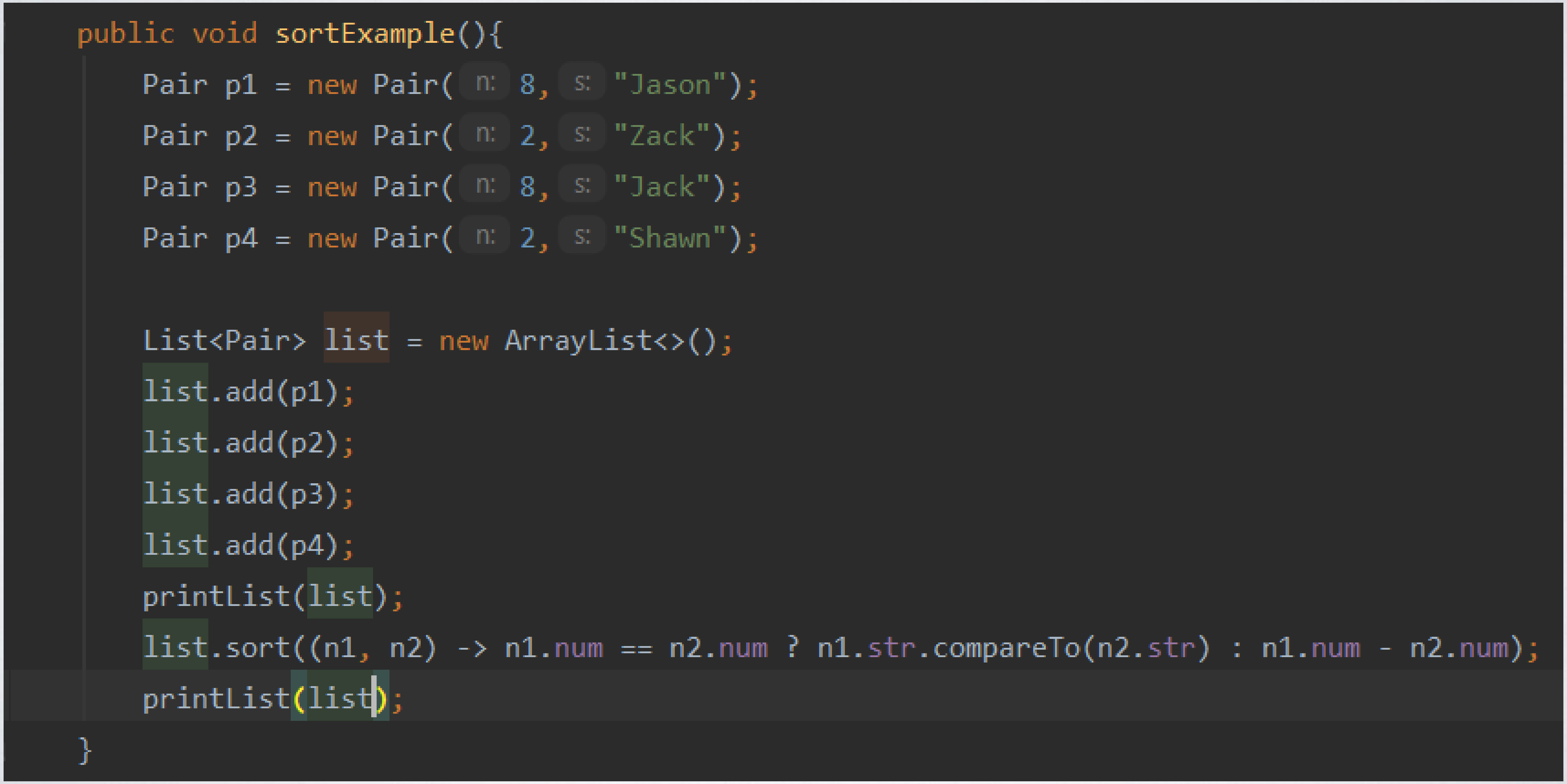
CUSTOM ORDERING

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

COLLECTIONS

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

Collections.sort() Example



```java
public void sortExample(){
    Pair p1 = new Pair( n: 8, s: "Jason");
    Pair p2 = new Pair( n: 2, s: "Zack");
    Pair p3 = new Pair( n: 8, s: "Jack");
    Pair p4 = new Pair( n: 2, s: "Shawn");

    List<Pair> list = new ArrayList<>();
    list.add(p1);
    list.add(p2);
    list.add(p3);
    list.add(p4);
    printList(list);
    list.sort((n1, n2) -> n1.num == n2.num ? n1.str.compareTo(n2.str) : n1.num - n2.num);
    printList(list);
}
```

QUESTIONS?