Collections

Summary

- You will know Collections
- You will recognize Set/List and their derivates
- We will know basic collection operations

SCANNER
I/O

https://docs.oracle.com/javase/tutorial/essential/io/index.h

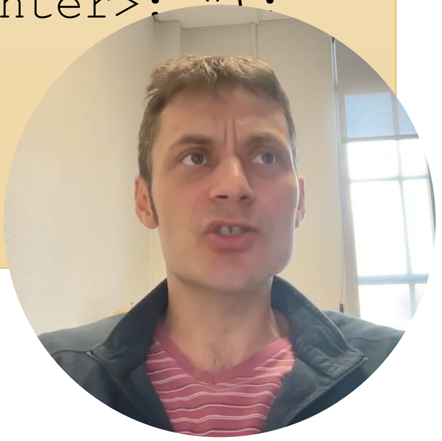
Scanning

- The scanner API breaks input into individual tokens associated with bits of data.

```
s = new Scanner(new File ("text.txt"));
s.hasNextDouble();
while (s.hasNext()) {
  //print
}
```

Read from Standard.in

```
import java.util.Scanner;

Scanner scanner = new Scanner(System.in);
System.out.print("Enter a line of text followed by <enter>: ");
String str = scanner.readline();

System.out.print("Enter an integer followed by <enter>: ");
int i = scanner.readInt();

System.out.print("Enter a floating-point number followed by <enter>: ");
double d = scanner.readDouble();
```



Read from File

```
import java.util.Scanner;
import java.io.File;

Scanner scanner = new Scanner(new File("ints.txt"));
while (scanner.hasNext()) {
        int i = scanner.nextInt();
        System.out.println(i);
}

scanner = new Scanner(new File("lines.txt"));
while (scanner.hasNext()) {
        String str = scanner.nextLine();
        System.out.println(str);
}
```

COLLECTIONS, STREAMS

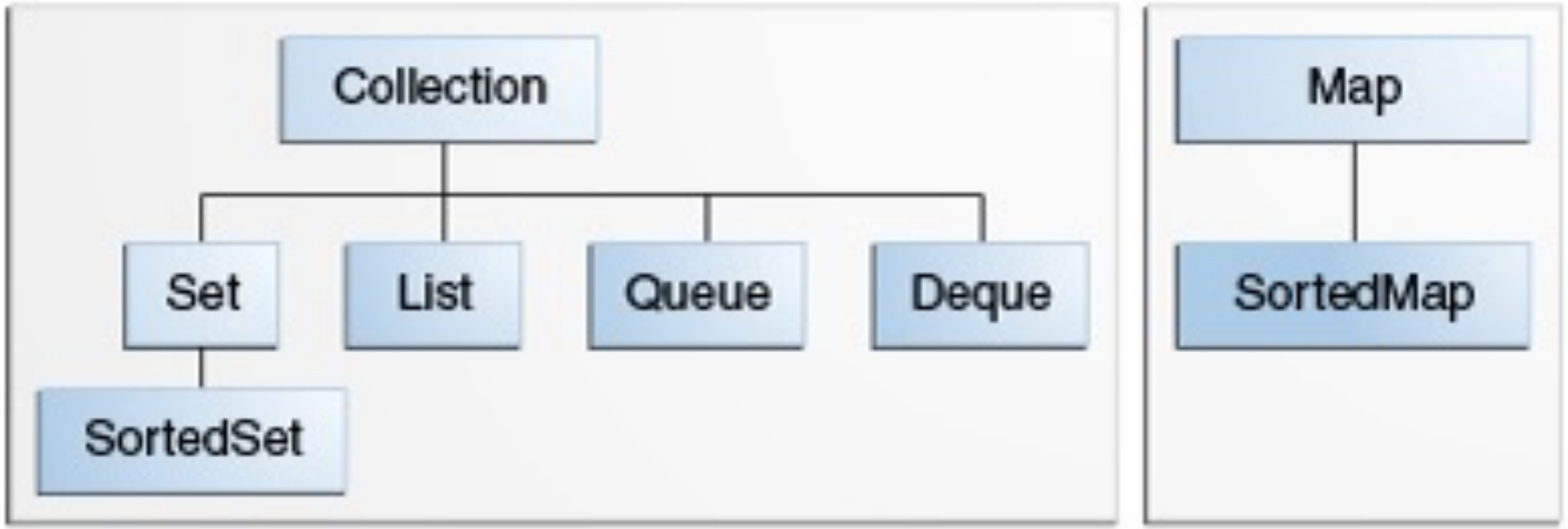
(NEW IN JDK 1.8)

https://docs.oracle.com/javase/tutorial/collections/intro/ind

Intro (its all about collections!)

- **Java Collections Framework**
  - interfaces, implementations, aggregate operations & algorithms
  - A *collection* — sometimes called a container — is simply an object that groups multiple elements into a single unit.
  - Collections are used to store, retrieve, manipulate, & com
aggregate data.

Hierarchy



https://docs.oracle.com/javase/tutorial/collections/intro/index.html

Hierarchy

- Collection data structures interfaces are **generic**.
- Guess why?

- Declaration of the Collection interface:
- `public interface Collection<E>...`

- The <E> syntax tells you that the interface is generic.

https://docs.oracle.com/javase/tutorial/collections/intro/index.html

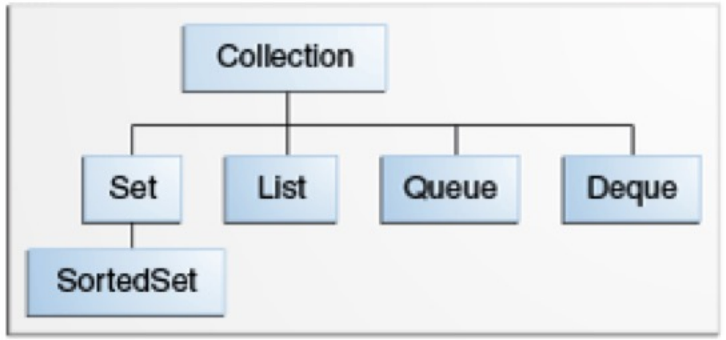
Generics

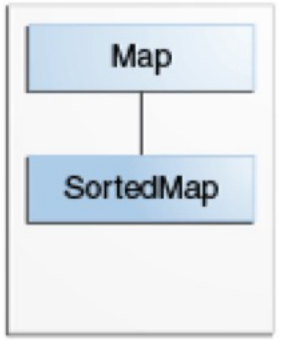
- Generics enable *types* (classes and interfaces) to be parameters when defining classes, interfaces, and methods.
- Benefits:
  - *Stronger **type checks** at compile time.*
  - *Elimination of casts.*

```
// need cast
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

```
// no cast needed
List<String> list = new ArrayList<String
list.add("hello");
String s = list.get(0); // no cast
```

Hierarchy





- Collection — the root of the collection hierarchy.
  - *A collection represents a group of objects known as its elements.*
  - *The Collection interface is the least common denominator that all collections implement.*
  - *It is used to pass collections around and to manipulate th maximum generality is desired.*

Hierarchy

– *Some types of collections allow duplicate elements, and others do not.*

- Examples?

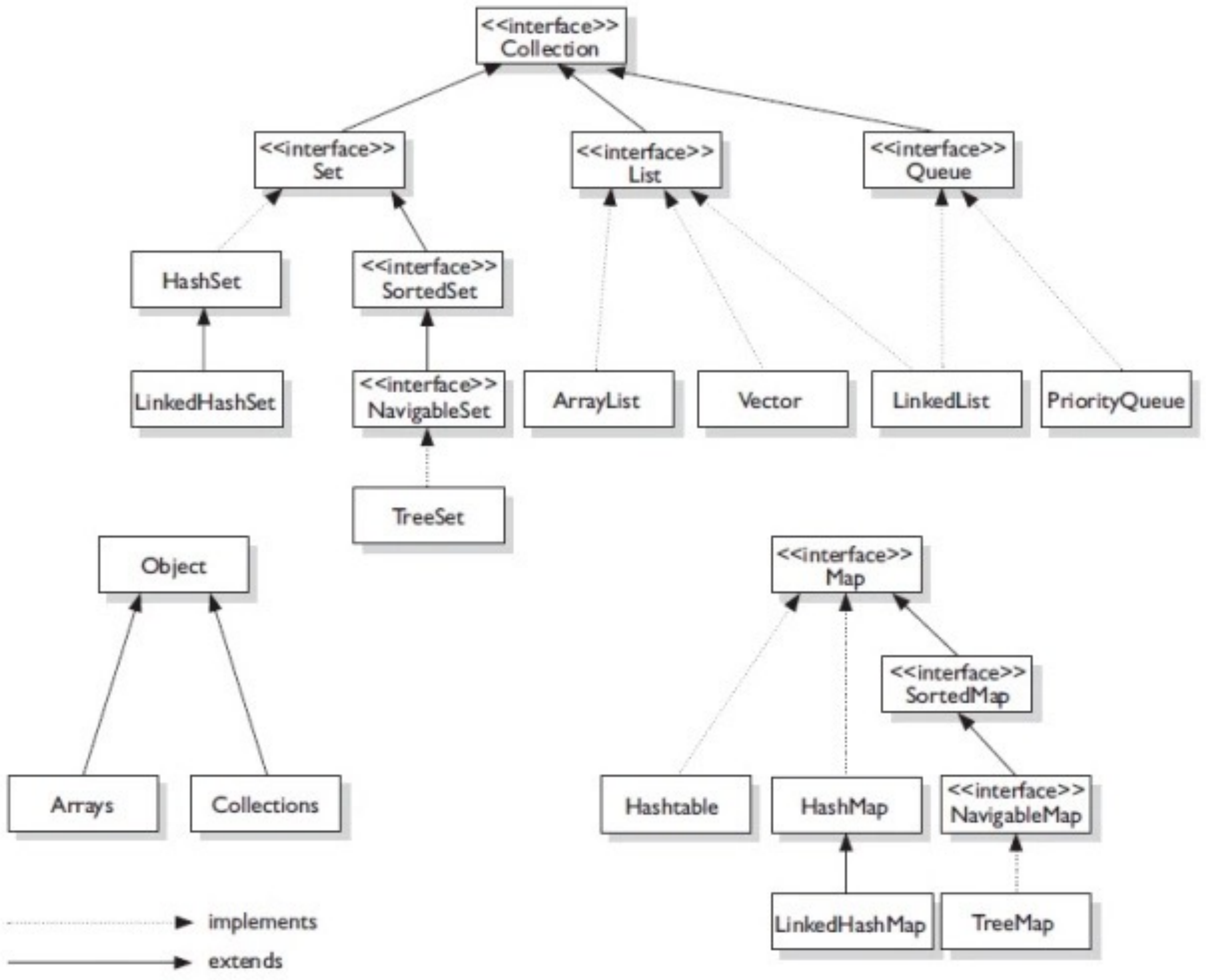
– *Some are ordered and others are unordered.*

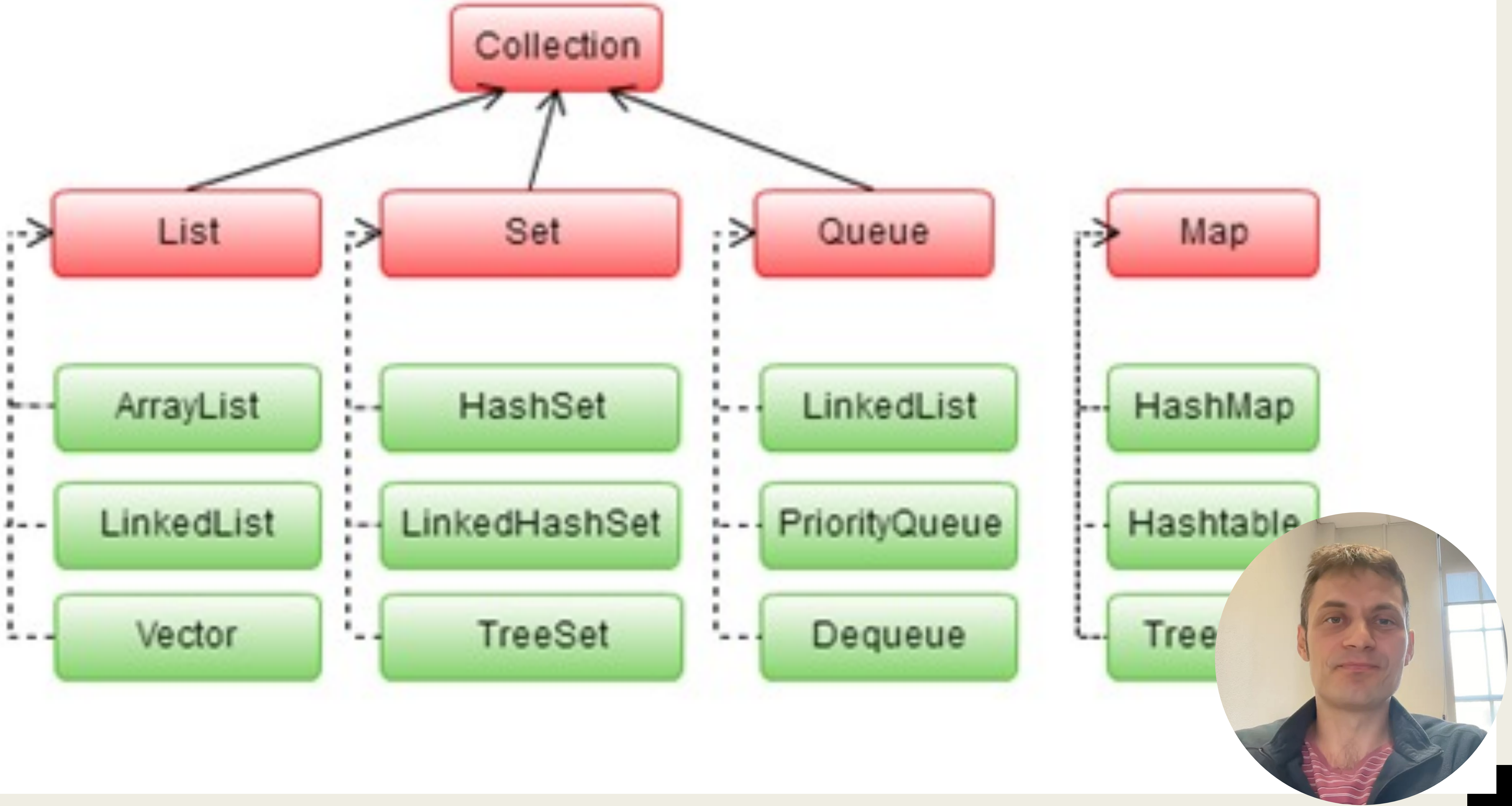
- Examples?

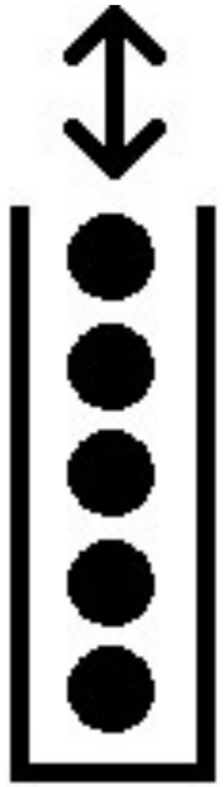
– *The Java platform doesn't provide any direct implementati interface but provides implementations of more specific subinterfaces, such as Set and List.*

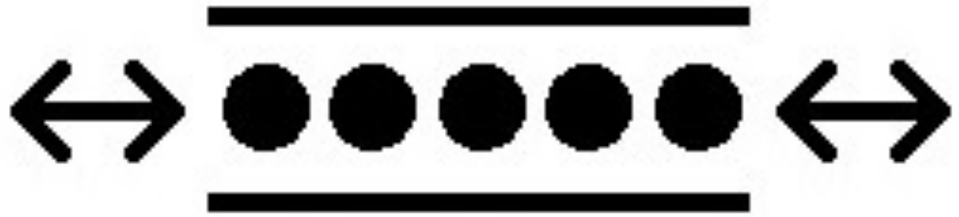
Hierarchy

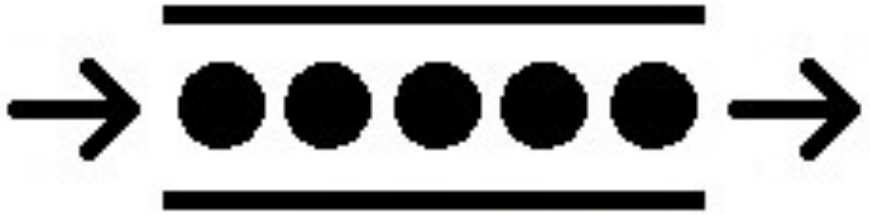
- Set
- List
- Map
- SortedSet
- SortedMap

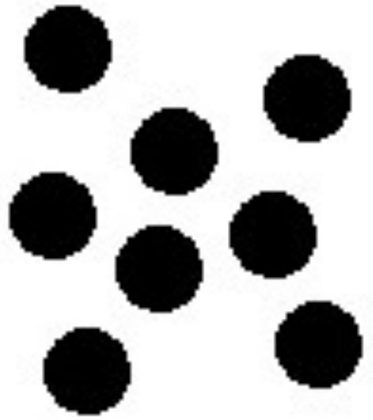

<<interface>> Collection
<<interface>> Set
<<interface>> List
<<interface>> Queue
HashSet
<<interface>> SortedSet
LinkedHashSet
<<interface>> NavigableSet
ArrayList
Vector
LinkedList
PriorityQueue
TreeSet
Object
Arrays
Collections
<<interface>> Map
<<interface>> SortedMap
Hashtable
HashMap
<<interface>> NavigableMap
LinkedHashMap
TreeMap
implements
extends

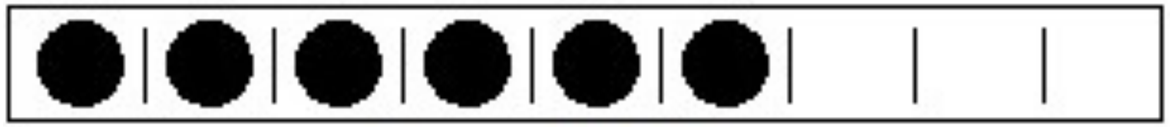

Collection
List
Set
Queue
Map
ArrayList
LinkedList
Vector
HashSet
LinkedHashSet
TreeSet
LinkedList
PriorityQueue
Dequeue
HashMap
Hashtable
Tree

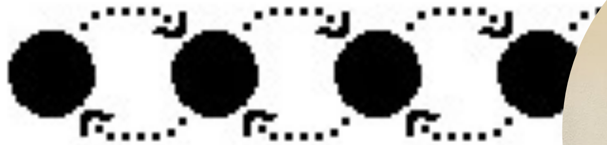

Stack


Deque


Queue


Collection


ArrayList


LinkedLis

Differences

- `List<String> list = List.of("Java", ″C++", ″Java");`
- `Set<String> set = Set.of("Java", ″C++", ″Java");`

- `Set<Integer> primes = Set.of(2, 3, 5, 7, 2);`
- `List <Integer> primesList = List.of(2, 3, 5, 7, 2);`

-
  ```
  Map<String, Integer> map = Map.of(
          ″one", 1,
          ″two", 2,
          ″three″, 3);
  ```

Hierarchy

- Set — a collection
- **Cannot contain duplicate elements.**

- This interface models the mathematical set abstraction and is used to represent sets:
  - *such as the cards comprising a poker hand,*
  - *the courses making up a student's schedule, or the*
  - *processes running on a machine (by default* ***no or***

Hierarchy

- List — an **ordered collection** (sometimes called a *sequence*).
- Lists can contain **duplicate elements.**

- The user of a List generally has precise control over where in the list each element is inserted and can access elements by their integer index (position).
- If you've used Vector, you're familiar with the genera of List.

Hierarchy

- Map — an object that **maps keys to values.**

  `Map<Key,Value>` e.g. 1 -> “one”

- A Map **cannot contain duplicate keys**; each key can map to at most one value.
- If you've used Hashtable, you're already familiar wi basics of Map.

Other interesting interfaces

- SortedSet — a Set that maintains its elements in **ascending order**. Several additional operations are provided to take advantage of the ordering. Sorted sets are used for naturally ordered sets, such as word lists and membership rolls.
- SortedMap — a Map that maintains its mappings in **ascending key order**. This is the Map analog of SortedSet. Sorted maps are used for naturally ordered collections of key/value pairs, such as dictionaries and telephone directories.

COLLECTION API

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/List.html

Usage

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

- The Collection interface contains methods that perform basic operations, such as
  - *int size(),*
  - *boolean isEmpty(),*
  - *boolean contains(Object element),*
  - *boolean add(E element),*
  - *boolean remove(Object element),*
  - *and Iterator<E> iterator().*

- It also contains methods that operate on entire collections, such as
  - *boolean containsAll(Collection<?> c),*
  - *boolean addAll(Collection<? extends E> c),*
  - *boolean removeAll(Collect*
  - *boolean retainAll(Collecti*
  - *and void clear().*

Recent news

- In JDK 8 and later, the Collection interface also exposes methods Stream<E> stream() and Stream<E> parallelStream(), for obtaining sequential or parallel streams from the underlying collection (greatly simplifies filtering and mapping via lambdas).

```
for (Person p : roster) {
        System.out.println(p.getName());
}
```

```
roster
  .stream().forEach(e ->
                System.out.println(e.g
```

Traversing collection

- (1) For-each
- (2) Iterator
- (3) Streams
  - *Next week*

Traversing collection

- (1) For-each

```
for (Object o : collection) {
        System.out.println(o);
}
```

Traversing collection

- (2) Iterator
  - an object that enables you to traverse through a collection and to remove elements from the collection selectively, if desired.

```
public interface Iterator<E> {
  boolean hasNext();
  E next();
  void remove(); //optional
}
```

```
static void filter(Collection<?> c) {
   for (Iterator<?> it = c.iterator(); it.hasNext(); ) {
            if (!cond(it.next())) {
                 it.remove();
            }
   }
}
```

Operations

- As a simple example of the power of bulk operations, consider the following idiom to remove *all* instances of a specified element, e, from a Collection, c.
  - *c.removeAll(Collections.singleton(e));*
- For example, suppose that c is a Collection. The following snippet dumps the contents of c into a newly allocated array of Object whose length is identical to the number of elements in c.
  - *Object[] a = c.toArray();*
- Suppose that c is known to contain only strings (perhaps because c i type Collection<String>). The following snippet dumps the conten a newly allocated array of String whose length is identical to the elements in c.
  - *String[] a = c.toArray(new String[0]);*

Next to explore in Java!

- Set
- List
- Map (Part 2)

Visualization here: https://visualgo.net/en

Set

- A Set is a Collection that cannot contain duplicate elements. It models the mathematical set abstraction. The Set interface contains *only* methods inherited from Collection and adds the restriction that duplicate elements are prohibited.
  - *Collection<Type> noDups = new HashSet<Type>(c);*

```
Set<String> set = new HashSet<>();
set.add("myPin");
if (set.contains("myPin")) {
  set.remove("myPin")
}
```

Task get rid of duplicates

```
public class FindDups {
      public static void main(String[] args) {
            Set<String> set = new HashSet<String>();
            for (String a : args) {
                  set.add(a);
            }
            System.out.println(set.size()+" distinct words:"+ set);
            }
}
```

```
java FindDups i came i saw i left Dr Cernys class with A
```

Set Operations

- To calculate the union, intersection, or set difference of two sets *nondestructively* (without modifying either set), the caller must copy one set before calling the appropriate bulk operation. The following are the resulting idioms.

```
Set<Type> union = new HashSet<Type>(s1);
union.addAll(s2);

Set<Type> intersection = new HashSet<Type>(s1);
intersection.retainAll(s2);

Set<Type> difference = new HashSet<Type>(s1);
difference.removeAll(s2);
```

Task get rid of duplicates and list them

```
public class FindDups2 {
  public static void main(String[] args) {
    Set<String> uniques = new HashSet<String>();
    ..
    for (String a : args)  {
      ..uniques add..
          ..
    }
    ..remove duplicates from all will leave uniques
} }
```

```
Unique words: [left, saw, came, Dr, Cernys, class, with, A]
Duplicate words: [i]
```

Task get rid of duplicates and list them

```
public class FindDups2 {
  public static void main(String[] args) {
    Set<String> uniques = new HashSet<String>();
    Set<String> dups = new HashSet<String>();
    for (String a : args)  {
      if (!uniques.add(a))
          dups.add(a);
    }
    uniques.removeAll(dups);  // Destructive set-difference
    System.out.println("Unique words: " + uniques);
    System.out.println("Duplicate words: " + dups);
} }
```

```
Unique words: [left, saw, came, Dr, Cernys, class, with, A]
Duplicate words: [i]
```

List

```
List<String> list = new ArrayList<>();
list.add("myPin");
list.get(0);
list.set(0,"myNewPin");
```

- A List is an ordered Collection (sometimes called a *sequence*). Lists may contain duplicate elements. In addition to the operations inherited from Collection, the List interface includes operations for the following:
- Positional access — manipulates elements based on their numerical position in the list. This includes methods such as get, set, add, addAll, and remove.
- Search — searches for a specified object in the list and returns its numerical position. Search methods include indexOf and lastIndexOf.
- Iteration — extends Iterator semantics to take advantage of the list's sequential nature. The listIterator methods provide this behavior.
- Range-view — The sublist method performs arbitrary *range opera* list.

List examples

- The add and addAll operations always append the new element(s) to the *end* of the list
  - *list1.addAll(list2);*

- Here's a nondestructive form of this idiom, which produces a third List consisting of the second list appended to the first.
  - *List<Type> list3 = new ArrayList<Type>(list1);*
  - *list3.addAll(list2);*

List positions

- The basic positional access operations are get, set, add and remove. Other operations (indexOf and lastIndexOf) return the first or last index of the specified element in the list.

```
public static <E> void swap(
               List<E> a, int i, int j) {
       E tmp = a.get(i);
       a.set(i, a.get(j));
       a.set(j, tmp);
}
```

```
public static void shuffle(
        List<?> list, Random rnd) {
   for (int i = list.size(); i > 1; i--) {
        swap(list, i - 1, rnd.nextInt
   }
}
```

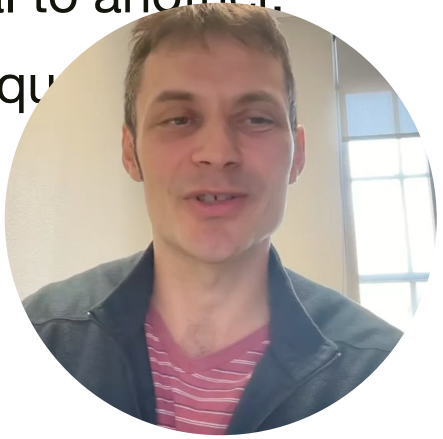
Example use

```
public class Shuffle {
        public static void main(String[] args) {
                List<String> list = Arrays.asList(args);
                Collections.shuffle(list);
                System.out.println(list);
        }
 }
```

Please explore on your own!

- Collections API
- Arrays API
- It is worth!
  - *Sort/Shuffle/Initiate/Convert/Reveres/Fill/Copy/Search/..*

- sort — sorts a List using a merge sort algorithm, which provides a fast, stable sort. (A *stable sort* is one that does not reorder equal elements.)
- shuffle — randomly permutes the elements in a List.
- reverse — reverses the order of the elements in a List.
- rotate — rotates all the elements in a List by a specified distance.
- swap — swaps the elements at specified positions in a List.
- replaceAll — replaces all occurrences of one specified value with another.
- fill — overwrites every element in a List with the specified value.
- copy — copies the source List into the destination List.
- binarySearch — searches for an element in an ordered List using the binary search algorithm.
- indexOfSubList — returns the index of the first sublist of one List that is equal to another.
- lastIndexOfSubList — returns the index of the last sublist of one List that is equ



Example Card Game using List

- *% java Deal 4 5*
- *[8 of hearts, jack of spades, 3 of spades, 4 of spades, king of diamonds]*
- *[4 of diamonds, ace of clubs, 6 of clubs, jack of hearts, queen of hearts]*
- *[7 of spades, 5 of spades, 2 of diamonds, queen of diamonds, 9 of clubs]*
- *[8 of spades, 6 of diamonds, ace of spades, 3 of hearts, ace of hearts]*

Sublist

The range-view operation, subList(int fromIndex, int toIndex), retu
this list whose indices range from fromIndex, inclusive, to toInde

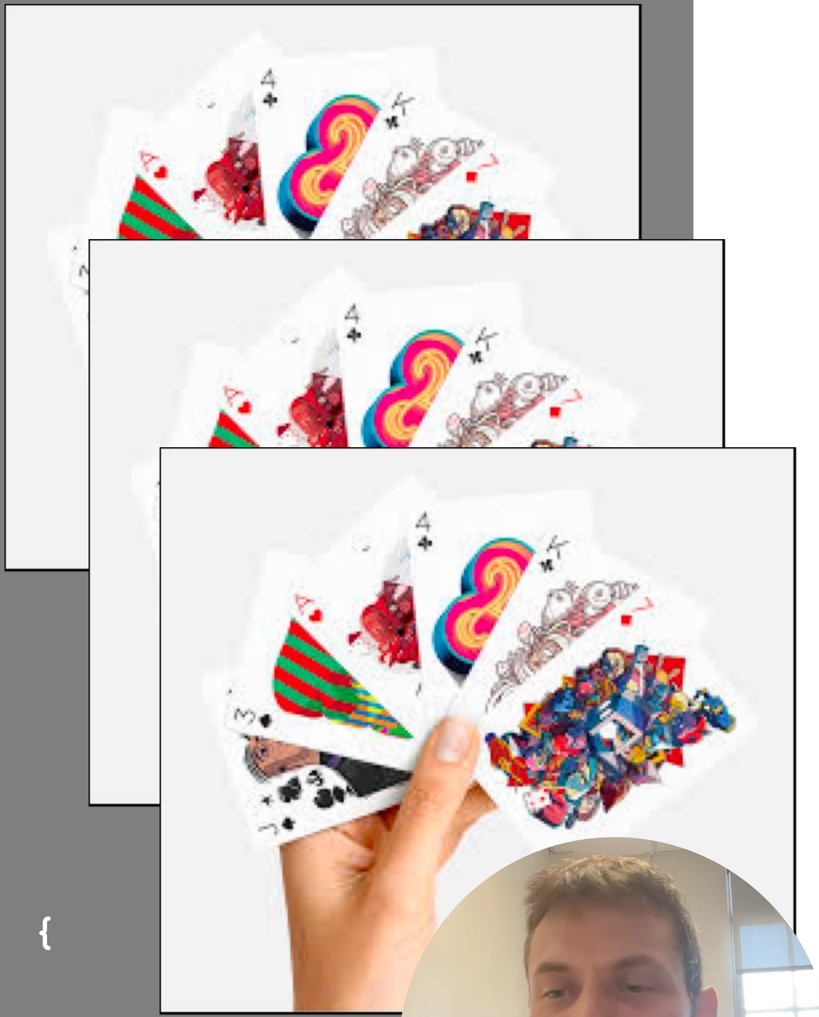
```
// removes a range of elements from a List.
list.subList(fromIndex, toIndex).clear();

int i = list.subList(fromIndex, toIndex).inde
int j = list.subList(fromIndex, toIndex).las   de
```

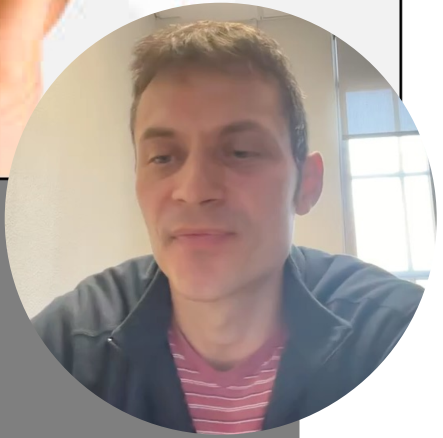
Returns a new List (the "hand") containing the specified number of elements taken from the end of the specified List (the "deck"). The elements returned in the hand are removed from the deck.

```
public static <E> List<E> dealHand(List<E> deck, int n) {
        int deckSize = deck.size();
        List<E> handView = deck.subList(deckSize - n, deckSize);
        List<E> hand = new ArrayList<E>(handView);
        handView.clear();
        return hand;
}
```

```
public class Deal {
 public static void main(String[] args) {
   if (args.length < 2) {
     System.err.println(”$Deal hands cards");
     return;
    }

  String[] suit = new String[] {
   "spades", "hearts", "diamonds", "clubs"
  };
  String[] rank = new String[] {
    "ace", "2", "3", “4", "5", "6", "7", "8", "9",
    "10", "jack", "queen", "king"
  };


  List<String> deck = new ArrayList<String>();
  for (int i = 0; i < suit.length; i++) {
     for (int j = 0; j < rank.length; j++) {
             deck.add(rank[j] + " of " + suit[i]);
          }
  }
… // deck of cards
 }
```

```
public class Deal {

 public static void main(String[] args) {
  .. // prepared the deck
  int numHands = Integer.parseInt(args[0]);
  int cardsPerHand = Integer.parseInt(args[1]);

  // Make a normal 52-card deck.
  if (numHands * cardsPerHand > deck.size()){
    System.err.println("Not enough cards.");
    return;
   }

  // Shuffle the deck.
  Collections.shuffle(deck);

  for (int i = 0; i < numHands; i++)
    out.println(dealHand(deck, cardsPerHand));
  }
}
public static <E> List<E> dealHand(List<E> deck, int n) {
  int deckSize = deck.size();
  List<E> handView = deck.subList(deckSize -n, deckSize);
  List<E> hand = new ArrayList<E>(handView);
  handView.clear();
  return hand;
}}
```





Example output

- *% java Deal 4 5*
- *[8 of hearts, jack of spades, 3 of spades, 4 of spades, king of diamonds]*
- *[4 of diamonds, ace of clubs, 6 of clubs, jack of hearts, queen of hearts]*
- *[7 of spades, 5 of spades, 2 of diamonds, queen of diamonds, 9 of clubs]*
- *[8 of spades, 6 of diamonds, ace of spades, 3 of hearts, ace of hearts]*