Java collections

Yulia Newton, Ph.D.

CS151, Object Oriented Programming

San Jose State University

Spring 2020

What are collections in Java?

- A group of individual objects represented by a single unit
- Java provides data structures geared towards
- Java collection classes
  - Collection interface *java.util.Collection*
  - Map interface *java.util.Map*
- Used to be achieved with arrays, vectors, hashtables, etc. before Java introduced Collection framework

Example: using array to store a list of integers

```
class Test
{
    public static void main (String[] args)
    {
        int myArray[] = new int[] {1, 2, 3, 4};
        System.out.println(myArray[0]);
        System.out.println(myArray[1]);
        System.out.println(myArray[2]);
        System.out.println(myArray[3]);
    }
}
```

Example: using Vector object to store a list of integers

```
import java.util.*;

class Test
{
    public static void main (String[] args)
    {
        Vector<Integer> myVector = new Vector();
        myVector.addElement(1);
        myVector.addElement(2);

        System.out.println(myVector.elementAt(0));
        System.out.println(myVector.elementAt(1));
    }
}
```

Example: using hashtable to store a list of mapped key-value pairs

```
import java.util.*;

class Test
{
    public static void main (String[] args)
    {
        Hashtable<Integer, String> myHash = new Hashtable();
        myHash.put(1,"hash1");
        myHash.put(2,"hash2");

        System.out.println(myHash.get(1));
        System.out.println(myHash.get(2));
    }
}
```

Disadvantages of using these data structures

- No consistent API
- No standard member access methods
- Need to know how to use each data structure
- Some are non-extendable/final classes
  - E.g. *Vector*

Java Collection framework

- Java provides predefined architecture for storing and manipulating groups of objects
  - A set of classes and interfaces
  - Methods and algorithms
- Commonly used data structures
- Can be thought of as a library of common data structures and functionality
- Reduces programming efforts
- Increases performance
- Promotes code re-use

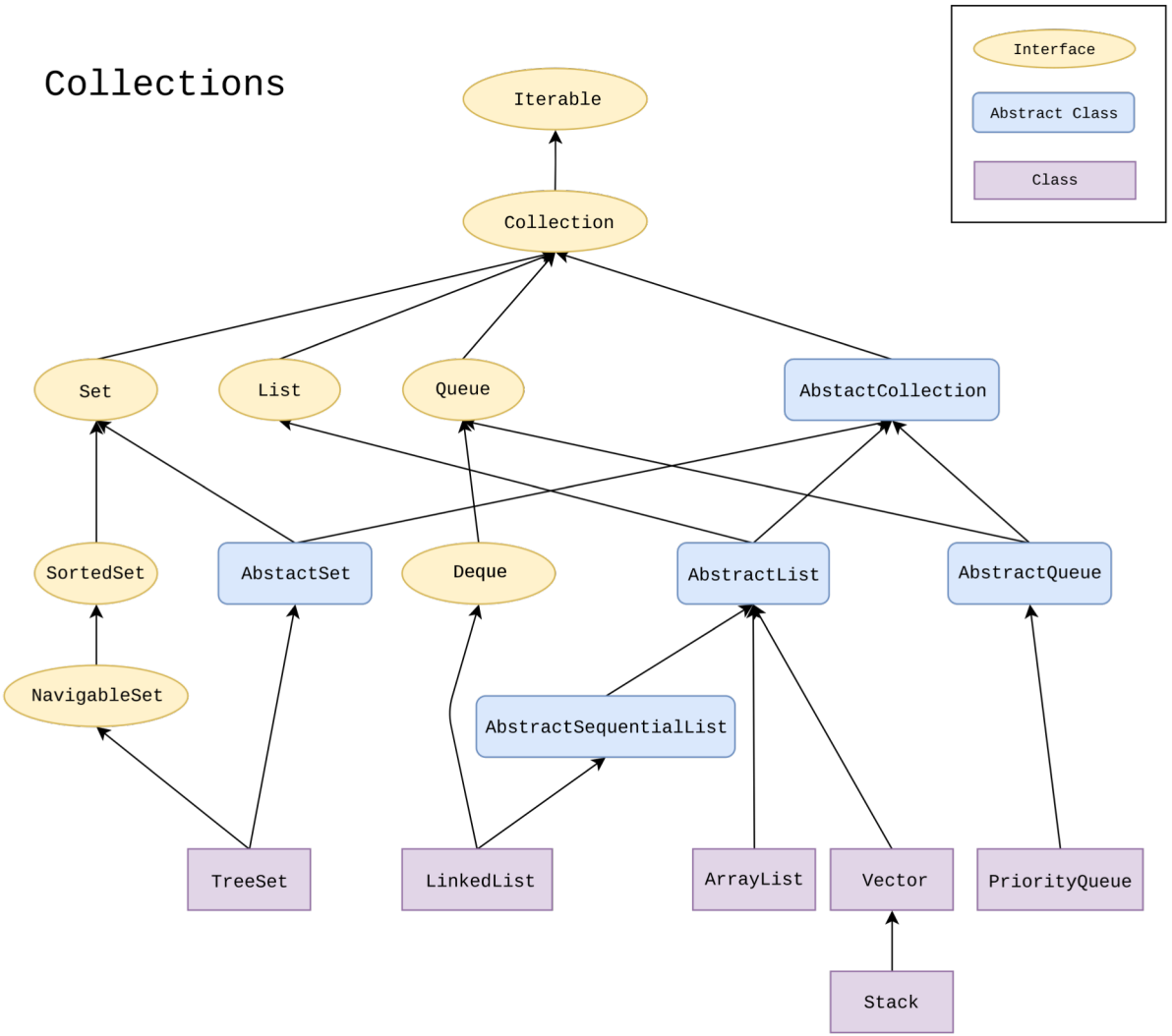
Types of collections

- Ordered lists
- Unordered lists
- Dictionaries/maps

Members of Java collection framework

- Java collection framework provides
  - Classes
    - *ArrayList*, *Vector*, *LinkedList*, *PriorityQueue*, *HashSet*, *LinkedHashSet*, *TreeSet*
  - Interfaces
    - *Set*, *List*, *Queue*, *Deque*

Java Collection framework



https://en.wikipedia.org/wiki/Java_collections_framework

Iterable interface

- Root interface for all collections in Java
- Directly extended by Collection interface

Collection interface

- Extends Iterable
- Directly extended by List, Queue, Set

https://docs.oracle.com/javase/9/docs/api/java/util/Collection.html

Collection methods

Modifier and Type	Method	Description
boolean	**add**(**E** e)	Ensures that this collection contains the specified
boolean	**addAll**(**Collection**<? extends **E**> c)	Adds all of the elements in the specified collection to this collection (op
void	**clear**()	Removes all of the elements from this collection (optional operation).
boolean	**contains**(**Object** o)	Returns `true` if this collection contains the specified element.
boolean	**containsAll**(**Collection**<?> c)	Returns `true` if this collection contains all of the elements in the specif
boolean	**equals**(**Object** o)	Compares the specified object with this collection for equality.
int	**hashCode**()	Returns the hash code value for this collection.
boolean	**isEmpty**()	Returns `true` if this collection contains no elements.
Iterator<**E**>	**iterator**()	Returns an iterator over the elements in this collection.
default **Stream**<**E**>	**parallelStream**()	Returns a possibly parallel `Stream` with this collection as its source.
boolean	**remove**(**Object** o)	Removes a single instance of the specified element from this collection
boolean	**removeAll**(**Collection**<?> c)	Removes all of this collection's elements that are also contained in the
default boolean	**removeIf**(**Predicate**<? super **E**> filter)	Removes all of the elements of this collection that satisfy the given pre
boolean	**retainAll**(**Collection**<?> c)	Retains only the elements in this collection that are contained in the sp
int	**size**()	Returns the number of elements in this collection.
default **Spliterator**<**E**>	**spliterator**()	Creates a **Spliterator** over the elements in this collection.
default **Stream**<**E**>	**stream**()	Returns a sequential `Stream` with this collection as its source.
Object[]	**toArray**()	Returns an array containing all of the elements in this collection.
<T> T[]	**toArray**(T[] a)	Returns an array containing all of the elements in this collection; the ru

List interface

- An ordered collection of objects
  - Sequence
- Used when the order and positions of elements in the list must be controlled
  - Access elements in the list by integer index
- Lists allow duplicates
- Directly implemented by *ArrayList*, *Vector*, and *LinkedList* classes
  - *Stack* extends *Vector*
- Instantiate as follows
  - *List <Type> myList1 =* ***new*** *ArrayList();*
  - *List <Type> myList2 =* ***new*** *LinkedList();*
  - *List <Type> myList3 =* ***new*** *Vector();*
  - *List <Type> myList4 =* ***new*** *Stack();*

ArrayList vs. Vector vs. LinkedList vs. Stack

- *ArrayList* uses dynamic array to store data, maintains the insertion order, allows random access, is non-synchronized
  - Synchronized = thread safe
- *Vector* is similar to *ArrayList* but is synchronized, contains additional methods
- *Stack* is a subclass of *Vector*, implements last-in-first-out structure (no random access), contains stack-specific methods
- *LinkedList* is internally a doubly-linked list, maintains insertion order, non-synchronized, manipulations of this data structure are usually fast

Example: working with a ArrayList of Strings

We need an iterator to iterate through list elements

Add an element at a specific index

Remove elements by index or by value

```
public class Test {
    public static void main(String[] args){
        List<String> names = new ArrayList<String>();

        // add elements:
        names.add("Jeff");
        names.add("Dan");
        names.add("Joe");
        names.add("Annie");
        names.add("Casey");
        names.add(2,"Anthony");

        // iterate over elements:
        Iterator nameIterator = names.iterator();
        while(nameIterator.hasNext()){
            System.out.println(nameIterator.next());
        }
        System.out.println();

        // access specific element by index:
        System.out.println("Element at index 1 is "+names.get(1));

        // test presence of elements in the list:
        System.out.println("names contains Joe ="+names.contains("Joe"));
        System.out.println("index of Casey is "+names.indexOf("Casey"));

        // remove some elements:
        names.remove(1);
        names.remove("Anthony");

        // convert List to array:
        String myArray[] = new String[names.size()];
        myArray = names.toArray(myArray);
        System.out.println("\nAfter converting to array:");
        for(String s : myArray){
            System.out.println(s);
        }
    }
}
```

Output:

```
Jeff
Dan
Anthony
Joe
Annie
Casey

Element at index 1 is Dan
names contains Joe =true
index of Casey is 5

After converting to array:
Jeff
Joe
Annie
Casey
```

Note: many more methods are available, refer to Java API

Example: vector of integers

Note: many more methods are available, refer to Java API

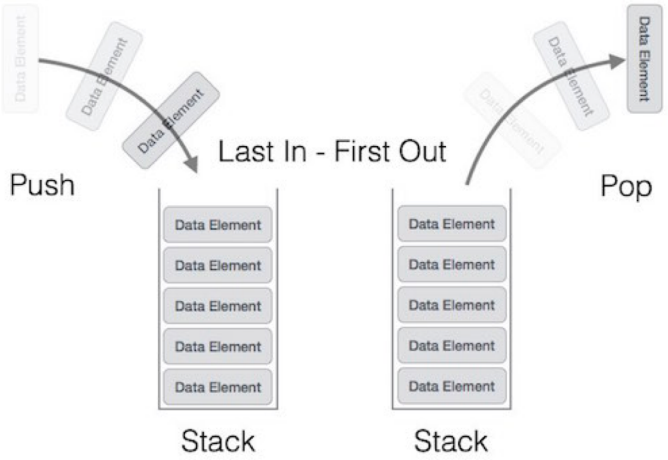
```java
public class Test {
    public static void main(String[] args){
        List<Integer> values = new Vector<Integer>();

        // add elements:
        values.add(1);
        values.add(5);
        values.add(2);
        values.add(7);
        values.add(10);
        values.add(2,6);

        // iterate over elements:
        Iterator valIterator = values.iterator();
        while(valIterator.hasNext()){
            System.out.println(valIterator.next());
        }
        System.out.println();

        // access specific element by index:
        System.out.println("Element at index 1 is "+values.get(1));

        // test presence of elements in the list:
        System.out.println("values contains 3 ="+values.contains(3));
        System.out.println("index of 7 is "+values.indexOf(7));

        // remove some elements:
        values.remove(1);

        // get sublist:
        System.out.println("Elements at indices 1 through 4: "+values.subList(1, 4));
    }
}
```

Output:

```
1
5
6
2
7
10

Element at index 1 is 5
values contains 3 =false
index of 7 is 4
Elements at indices 1 through 4: [6, 2, 7]
```

Stack data structure

- LIFO data structure
  - Last in first out
    - Elements are added at the end of the queue
    - Elements are removed from the beginning of the queue
- Stack data structure has specialized methods: push(), peek(), pop()



https://www.tutorialspoint.com/

Example: stack of integers

```java
public class Test {
    public static void main(String[] args){
        Stack<Integer> values = new Stack<Integer>();

        // add elements:
        values.push(1);
        values.push(5);
        values.push(2);
        values.push(7);
        values.push(10);
        values.push(6);

        // iterate over elements:
        Iterator valIterator = values.iterator();
        while(valIterator.hasNext()){
            System.out.println(valIterator.next());
        }
        System.out.println();

        // peek at the top element:
        System.out.println("Top element is "+values.peek());

        // search stack for an element:
        System.out.println("Index of value 10 is "+values.search(10));

        // remove the top element:
        values.pop();
        System.out.println("After popping top element: "+values);

        // Vector methods still work on a Stack data structure:
        values.remove(1);
        System.out.println("After removing element at index 1: "+values);
    }
}
```

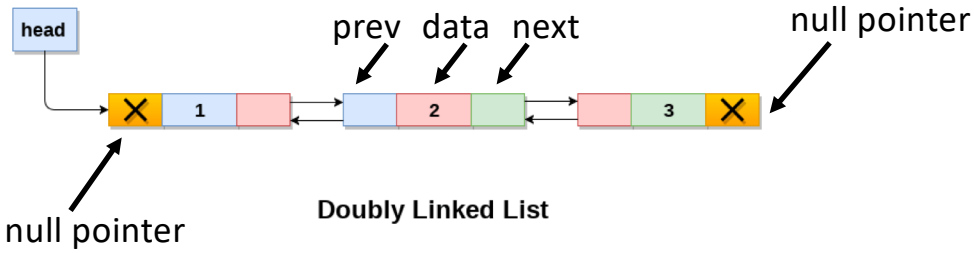
Declare as *Stack* because we want to use methods not available to *List* objects

Output:

```
1
5
2
7
10
6

Top element is 6
Index of value 10 is 2
After popping top element: [1, 5, 2, 7, 10]
After removing element at index 1: [1, 2, 7, 10]
```

Linked lists

- Linear data structure where elements are not stored in contiguous allocated location
- We have to traverse the linked list in order to access elements
- Elements are linked by pointers to the next location
- Doubly linked lists have pointers to the previous location
- Java LinkedList is implemented with doubly linked lists



https://www.javatpoint.com

Linked lists (cont'd)

- When an element of a doubly linked list is deleted:
  - next pointer of the previous element has to be pointed to the next element of the deleted item
  - previous pointer of the next element has to be pointed to the previous element of the deleted item
- *LinkedList* data structure can be used to represent a *Stack*

Example: you can do many of the same manipulations with LinkedList as with ArrayList

```
public class Test {
    public static void main(String[] args){
        List<String> names = new LinkedList<String>();

        // add elements:
        names.add("Jeff");
        names.add("Dan");
        names.add("Joe");
        names.add("Annie");
        names.add("Casey");
        names.add(2,"Anthony");

        // iterate over elements:
        Iterator nameIterator = names.iterator();
        while(nameIterator.hasNext()){
            System.out.println(nameIterator.next());
        }
        System.out.println();

        // access specific element by index:
        System.out.println("Element at index 1 is "+names.get(1));

        // test presence of elements in the list:
        System.out.println("names contains Joe ="+names.contains("Joe"));
        System.out.println("index of Casey is "+names.indexOf("Casey"));

        // remove some elements:
        names.remove(1);
        names.remove("Anthony");
    }
}
```

we declare *LinkedList* instead o *ArrayList*

Example: additional methods in *LinkedList* class

```
public class Test {
    public static void main(String[] args){
        LinkedList<String> names = new LinkedList<String>();

        // add elements:
        names.add("Jeff");
        names.add("Dan");
        names.add("Joe");
        names.add("Annie");
        names.add("Casey");
        names.add(2,"Anthony");
        names.addFirst("Tanya");
        names.addLast("Teddy");

        // additional methods available in LinkedList class:
        System.out.println("Head element, peek(): "+names.peek());
        System.out.println("First element, peekFirst(): "+names.peekFirst());
        System.out.println("Last element, peekLast(): "+names.peekLast());
        System.out.println("Retreve and remove head element, pollFirst(): "+names.pollFirst());
        System.out.println("Retreve and remove tail element, pollLast(): "+names.pollLast());

        // replaces the element at the specified index:
        names.set(2, "Brandy");

        // iterate over elements:
        Iterator nameIterator = names.iterator();
        while(nameIterator.hasNext()){
            System.out.println(nameIterator.next());
        }
    }
}
```

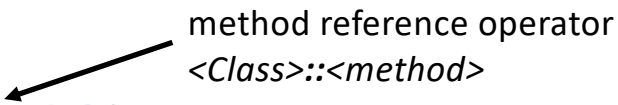
Note: many more methods are available, refer to Java API

Ways to iterate over elements of Java collections

- *Iterator* interface
  - *Iterator myIterator = myCollection.iterator();*
  - *while(iterator.hasNext()){…}*
- *ListIterator* interface
  - *To do*
- For-each loop
  - *for(Type element : myCollection){…}*
- For loop
  - To do
- *forEach()* method
  - To do
- *forEachRemaining()* method
  - To do

Example: iterating over elements of Java collection

```java
public class Test {
    public static void main(String[] args){
        List<String> names = new ArrayList<String>();
        names.add("Jeff");
        names.add("Dan");
        names.add("Joe");

        // Iterator interface:
        Iterator nameIterator = names.iterator();
        while(nameIterator.hasNext()){
            System.out.println(nameIterator.next());
        }
        System.out.println();

        // ListIterator interface:
        ListIterator<String> nameListIterator = names.listIterator();
        while( nameListIterator.hasNext() ){
            System.out.println(nameListIterator.next());
        }
        System.out.println();

        // for-each loop:
        for(String s:names){
            System.out.println(s);
        }
        System.out.println();

        // for loop:
        for(int i=0; i < names.size(); i++){
            System.out.println(names.get(i));
        }
        System.out.println();

        // forEach() method:
        names.forEach(System.out::println);
        System.out.println();

        // forEachRemaining() method:
        Iterator nameIterator2 = names.iterator();
        nameIterator2.forEachRemaining(System.out::println);
    }
}
```

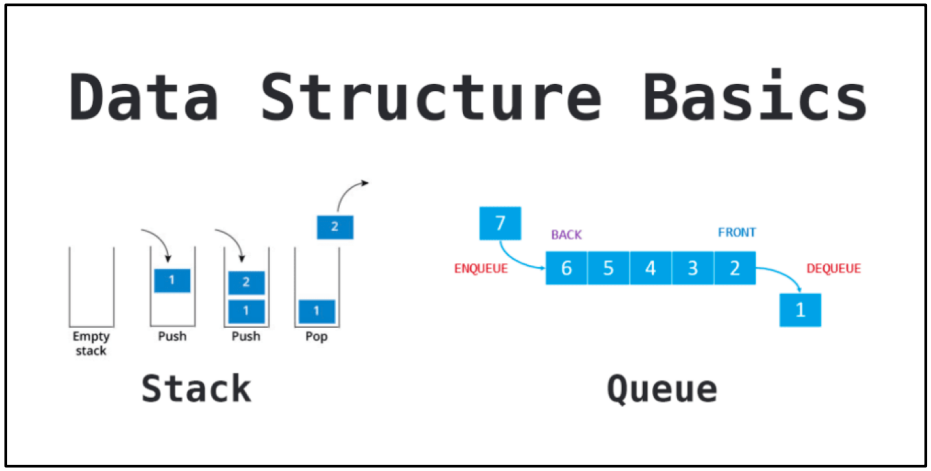


Queues

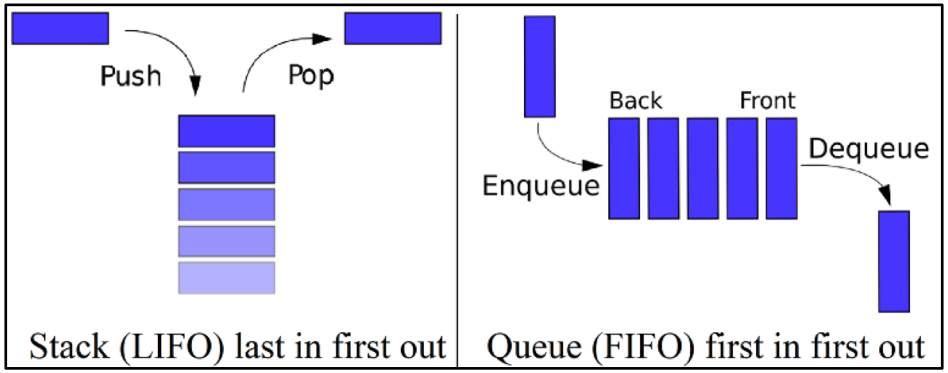
- FIFO data structure
  - First in first out
    - Opened from both ends
      - Elements are added at the rear of the queue
      - Elements are removed from the front of the queue
    - Similar to the way queues work in real life (e.g. bank)

www.thecrazyprogrammer.com

Queue vs. stack data structure



https://dev.to/rinsama77/data-structure-stack-and-queue-4ecd



https://dunglai.github.io/2018/07/26/linked-list/

Queue interface

- *Queue* interface extends *Collection* interface
- Directly implemented by *PriorityQueue* class and inherited by *Dequeue* interface
  - *ArrayDeque* class implements *Dequeue* interface

Summary of Queue methods

	Throws exception	*Returns special value*
Insert	`add(e)`	`offer(e)`
Remove	`remove()`	`poll()`
Examine	`element()`	`peek()`

www.docs.oracle.com

PriorityQueue class

- Implements *Queue* interface
- Objects in priority queue are processed by their priorities
- Does not allow *null* values to be stored in this data structure
- Allows duplicate elements
  - Can implement your own class that extends *PriorityQueue* that will scan for duplicates
- Insertions are performed in the order, as regular queues
  - Enqueue
- Deletions are performed by the priority order
  - Element with the highest priority is removed first
  - Dequeue

Example: basic operations with *PriorityQueue*

```java
public class Test {
    public static void main(String[] args){
        PriorityQueue<String> names = new PriorityQueue<String>();

        // queue elements:
        names.add("Jeff");
        names.add("Dan");
        names.add("Joe");
        names.add("Annie");
        names.add("Casey");

        // iterate over elements:
        Iterator nameIterator = names.iterator();
        while(nameIterator.hasNext()){
            System.out.println(nameIterator.next());
        }
        System.out.println("");

        // additional methods available in LinkedList class:
        System.out.println("Head element, peek(): "+names.peek());

        // remove element from the front of the queue:
        names.poll();

        // dequeue elements:
        System.out.println("After polling the front of the queue:");
        while (!names.isEmpty()){
            System.out.println(names.remove());
        }
    }
}
```

The element at the front of the queue is the one with the highest priority (in this case, highest value String)

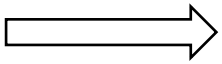
Output:

```
Annie
Casey
Joe
Jeff
Dan

Head element, peek(): Annie
After polling the front of the queue:
Casey
Dan
Jeff
Joe
```

dequeues all elements and empties the queue

Example: adding elements to the *PriorityQueue*

```java
public class Test {
    public static void main(String[] args){
        PriorityQueue<String> names = new PriorityQueue<String>();

        // queue elements:
        names.add("Jeff");
        Iterator nameIterator = names.iterator();
        while(nameIterator.hasNext()){
            System.out.println(nameIterator.next());
        }
        System.out.println("");

        names.add("Dan");
        nameIterator = names.iterator();
        while(nameIterator.hasNext()){
            System.out.println(nameIterator.next());
        }
        System.out.println("");

        names.add("Joe");
        nameIterator = names.iterator();
        while(nameIterator.hasNext()){
            System.out.println(nameIterator.next());
        }
        System.out.println("");

        names.add("Annie");
        nameIterator = names.iterator();
        while(nameIterator.hasNext()){
            System.out.println(nameIterator.next());
        }
        System.out.println("");

        names.add("Casey");
        nameIterator = names.iterator();
        while(nameIterator.hasNext()){
            System.out.println(nameIterator.next());
        }
        System.out.println("");

        names.add("Anthony");
        nameIterator = names.iterator();
        while(nameIterator.hasNext()){
            System.out.println(nameIterator.next());
        }
        System.out.println("");

        names.add("Alan");
        nameIterator = names.iterator();
        while(nameIterator.hasNext()){
            System.out.println(nameIterator.next());
        }
        System.out.println("");
    }
}
```



```
Jeff

Dan
Jeff

Dan
Jeff
Joe

Annie
Dan
Joe
Jeff

Annie
Casey
Joe
Jeff
Dan

Annie
Casey
Anthony
Jeff
Dan
Joe

Alan
Casey
Annie
Jeff
Dan
Joe
Anthony
```

Observe as the front of the elements at the front of the queue change

Example: poll front element to see how the front value changes

```java
public class Test {
    public static void main(String[] args){
        PriorityQueue<String> names = new PriorityQueue<String>();

        // queue elements:
        names.add("Jeff");
        names.add("Dan");
        names.add("Joe");
        names.add("Annie");
        names.add("Casey");
        names.add("Anthony");
        names.add("Alan");
        Iterator nameIterator = names.iterator();
        while(nameIterator.hasNext()){
            System.out.println(nameIterator.next());
        }
        System.out.println("");

        names.poll();
        nameIterator = names.iterator();
        while(nameIterator.hasNext()){
            System.out.println(nameIterator.next());
        }
        System.out.println("");
    }
}
```

Same example as before but we poll the front of the queue, which removes the front element

Output:

```
Alan
Casey
Annie
Jeff
Dan
Joe
Anthony

Annie
Casey
Anthony
Jeff
Dan
Joe
```

Priority queues with custom comparators

- Custom comparators allow customizing how priorities are assigned to elements in the priority queue
- *java.util.Comparator* interface

```
Comparator<String> myComparator = new Comparator<String>() {
    @Override
    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
};
```

Example: adding elements to the *PriorityQueue* with custom *Comparator*

```java
public class Test {
    public static void main(String[] args){
        Comparator<String> myComparator = new Comparator<String>() {
            @Override
            public int compare(String s1, String s2) {
                return s1.length() - s2.length();
            }
        };

        PriorityQueue<String> names = new PriorityQueue<String>(myComparator);

        // queue elements:
        names.add("Jeff");
        Iterator nameIterator = names.iterator();
        while(nameIterator.hasNext()){
            System.out.println(nameIterator.next());
        }
        System.out.println("");

        names.add("Dan");
        nameIterator = names.iterator();
        while(nameIterator.hasNext()){
            System.out.println(nameIterator.next());
        }
        System.out.println("");

        names.add("Joe");
        nameIterator = names.iterator();
        while(nameIterator.hasNext()){
            System.out.println(nameIterator.next());
        }
        System.out.println("");

        names.add("Annie");
        nameIterator = names.iterator();
        while(nameIterator.hasNext()){
            System.out.println(nameIterator.next());
        }
        System.out.println("");

        names.add("Casey");
        nameIterator = names.iterator();
        while(nameIterator.hasNext()){
            System.out.println(nameIterator.next());
        }
        System.out.println("");

        names.add("Anthony");
        nameIterator = names.iterator();
        while(nameIterator.hasNext()){
            System.out.println(nameIterator.next());
        }
        System.out.println("");

        names.add("Alan");
        nameIterator = names.iterator();
        while(nameIterator.hasNext()){
            System.out.println(nameIterator.next());
        }
        System.out.println("");
    }
}
```

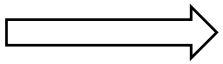
custom Comparator that compares strings based on their length

```
Jeff

Dan
Jeff

Dan
Jeff
Joe

Dan
Jeff
Joe
Annie

Dan
Jeff
Joe
Annie
Casey

Dan
Jeff
Joe
Annie
Casey
Anthony

Dan
Jeff
Joe
Annie
Casey
Anthony
Alan
```

Example: longest strings have the highest priority

```
Comparator<String> myComparator = new Comparator<String>() {
    @Override
    public int compare(String s1, String s2) {
        return s2.length() - s1.length();
    }
};
```

switch s1 and s2 in the previous example



```
Jeff

Jeff
Dan

Jeff
Dan
Joe

Annie
Jeff
Joe
Dan

Annie
Casey
Joe
Dan
Jeff

Anthony
Casey
Annie
Dan
Jeff
Joe

Anthony
Casey
Annie
Dan
Jeff
Joe
Alan
```

Example: *PriorityQueue* with Employee objects

```
class Employee{
    private String name;
    private double performanceRank;

    Employee(String name, double rank){
        this.name = name;
        this.performanceRank = rank;
    }

    public double getRank(){return this.performanceRank;}

    @Override
    public String toString(){
        return this.name+" ("+this.performanceRank+")";
    }
}
```

Employees are ranked by performance

Custom comparator compares employees based on their performance score

```
public class Test {
    public static void main(String[] args){
        Comparator<Employee> myComparator = new Comparator<Employee>() {
            @Override
            public int compare(Employee e1, Employee e2) {
                return (int) Math.floor(e2.getRank() - e1.getRank());
            }
        };

        PriorityQueue<Employee> employees = new PriorityQueue<Employee>(myComparator);

        // queue elements:
        employees.add(new Employee("Jeff", 1.1));
        employees.add(new Employee("Dan", 1.9));
        employees.add(new Employee("Joe", 0.5));
        employees.add(new Employee("Annie", 2.7));
        employees.add(new Employee("Casey", 3.1));
        employees.add(new Employee("Anthony", 1.6));
        employees.add(new Employee("Alan", 2.5));

        Iterator nameIterator = employees.iterator();
        while(nameIterator.hasNext()){
            System.out.println(nameIterator.next());
        }
    }
}
```

Output:

```
Casey (3.1)
Annie (2.7)
Alan (2.5)
Jeff (1.1)
Dan (1.9)
Joe (0.5)
Anthony (1.6)
```

Example: PriorityQueue of Animal objects with custom comparator based on # of legs

```
interface Animal{
    public abstract int getLegs();
}

class Dog implements Animal{
    private int numLegs;

    Dog(int legs){this.numLegs = legs;}
    public String toString(){return "I am a dog with "+this.numLegs+" legs";}
    public int getLegs(){return this.numLegs;}
}

class Cat implements Animal{
    private int numLegs;

    Cat(int legs){this.numLegs = legs;}
    public String toString(){return "I am a cat with "+this.numLegs+" legs";}
    public int getLegs(){return this.numLegs;}
}

class Tarantula implements Animal{
    private int numLegs;

    Tarantula(int legs){this.numLegs = legs;}
    public String toString(){return "I am a tarantula with "+this.numLegs+" legs";}
    public int getLegs(){return this.numLegs;}
}

class Shrimp implements Animal{
    private int numLegs;

    Shrimp(int legs){this.numLegs = legs;}
    public String toString(){return "I am a shrimp with "+this.numLegs+" legs";}
    public int getLegs(){return this.numLegs;}
}
```

```
public class Test {
    public static void main(String[] args){
        Comparator<Animal> myComparator = new Comparator<Animal>() {
            @Override
            public int compare(Animal a1, Animal a2) {
                return a2.getLegs() - a1.getLegs();
            }
        };

        PriorityQueue<Animal> animals = new PriorityQueue<Animal>(myComparator);

        // queue elements:
        animals.add(new Dog(4));
        animals.add(new Cat(4));
        animals.add(new Shrimp(10));
        animals.add(new Tarantula(8));
        animals.add(new Tarantula(7));
        animals.add(new Shrimp(9));

        Iterator nameIterator = animals.iterator();
        while(nameIterator.hasNext()){
            System.out.println(nameIterator.next());
        }
    }
}
```

any object that implements Animal interface can be collected in this priority queue

these two animals lost a leg in an accident

Output:

```
I am a shrimp with 10 legs
I am a tarantula with 8 legs
I am a shrimp with 9 legs
I am a cat with 4 legs
I am a tarantula with 7 legs
I am a dog with 4 legs
```

Alternatively can implement *Comparable* interface and override *compareTo()* method

- There must be a way to compare the objects you are storing in *PriorityQueue*
- One way is to have the object type implement *Comparable* interface
  - Override *compareTo()* method

Example: alternative implementation of *PriorityQueue* with Employee objects

```java
class Employee implements Comparable<Employee>{
    private String name;
    private double performanceRank;

    Employee(String name, double rank){
        this.name = name;
        this.performanceRank = rank;
    }

    public double getRank(){return this.performanceRank;}

    @Override
    public String toString(){
        return this.name+" ("+this.performanceRank+")";
    }

    public int compareTo(Employee employee) {
        if(this.performanceRank > employee.getRank()) {
            return -1;
        } else if (this.performanceRank < employee.getRank()){
            return 1;
        } else {
            return 0;
        }
    }
}
```

Class Employee implements *Comparable*

Implement *compareTo()*

No need for custom comparator here

```java
public class Test {
    public static void main(String[] args){
        PriorityQueue<Employee> employees = new PriorityQueue<Employee>();

        // queue elements:
        employees.add(new Employee("Jeff", 1.1));
        employees.add(new Employee("Dan", 1.9));
        employees.add(new Employee("Joe", 0.5));
        employees.add(new Employee("Annie", 2.7));
        employees.add(new Employee("Casey", 3.1));
        employees.add(new Employee("Anthony", 1.6));
        employees.add(new Employee("Alan", 2.5));

        Iterator nameIterator = employees.iterator();
        while(nameIterator.hasNext()){
            System.out.println(nameIterator.next());
        }
    }
}
```

ArrayDeque class

- *Deque* interface
  - Extends *Queue* interface
  - Linear collection
  - Allows inserting and removing elements to/from both ends of the queue
    - Deque stands for double-ended queue

- *ArrayDeque* class
  - Implements *Deque* interface and extends *AbstractCollection*
  - Faster than *ArrayList* and *Stack* classes
  - No capacity restrictions

Example: *ArrayDeque* of strings

```java
public class Test {
    public static void main(String[] args){
        ArrayDeque<String> names = new ArrayDeque<String>();

        // queue elements:
        names.add("Jeff");
        names.add("Dan");
        names.add("Joe");
        names.add("Annie");
        names.add("Casey");
        names.add("Anthony");
        names.add("Alan");

        Iterator nameIterator = names.iterator();
        while(nameIterator.hasNext()){
            System.out.println(nameIterator.next());
        }

        // retrieve but do not remove elements:
        names.getFirst();
        System.out.println("After getFirst(): "+names);
        names.getLast();
        System.out.println("After getLast(): "+names);

        // poll the front of the queue:
        names.poll();
        System.out.println("After poll(): "+names);
        names.pollFirst();
        System.out.println("After pollFirst(): "+names);

        // poll the end of the queue:
        names.pollLast();
        System.out.println("After pollLast(): "+names);
    }
}
```

Output:

```
Jeff
Dan
Joe
Annie
Casey
Anthony
Alan
After getFirst(): [Jeff, Dan, Joe, Annie, Casey, Anthony, Alan]
After getLast(): [Jeff, Dan, Joe, Annie, Casey, Anthony, Alan]
After poll(): [Dan, Joe, Annie, Casey, Anthony, Alan]
After pollFirst(): [Joe, Annie, Casey, Anthony, Alan]
After pollLast(): [Joe, Annie, Casey, Anthony]
```

Set interface

- *Set* interface extends *Collection* interface
- Directly implemented by *HashSet* and *LinkedHashSet* classes
  - Directly extended by *SortedSet* interface, which is implemented by *TreeSet* class
  - These classes are non-synchronized
- Represents unordered collection of objects
- Does not allow duplicate objects
- Allows at most one null value

Classes that represent sets

- *HashSet*
  - Uses hash table for storage
- *LinkedHashSet*
  - Implemented as a linked list
  - Maintains insertion order
- *TreeSet*
  - Uses a tree as storage
  - Ordered set
  - Elements are arranged in ascending order

Example: *HashSet* of names

```
public class Test {
    public static void main(String[] args){
        HashSet<String> names = new HashSet<String>();

        // add elements:
        names.add("Jeff");
        names.add("Dan");
        names.add("Jeff");
        names.add("Joe");
        names.add("Annie");
        names.add("Anthony");
        names.add("Annie");

        Iterator nameIterator = names.iterator();
        while(nameIterator.hasNext()){
            System.out.println(nameIterator.next());
        }
    }
}
```

Output:

```
Dan
Joe
Anthony
Jeff
Annie
```

Example: *LinkedListSet* of names

```
public class Test {
    public static void main(String[] args){
        LinkedHashSet<String> names = new LinkedHashSet<String>();

        // add elements:
        names.add("Jeff");
        names.add("Dan");
        names.add("Jeff");
        names.add("Joe");
        names.add("Annie");
        names.add("Anthony");
        names.add("Annie");

        Iterator nameIterator = names.iterator();
        while(nameIterator.hasNext()){
            System.out.println(nameIterator.next());
        }
    }
}
```

Output:

```
Jeff
Dan
Joe
Annie
Anthony
```

Example: *TreeSet* of names

```
public class Test {
    public static void main(String[] args){
        TreeSet<String> names = new TreeSet<String>();

        // add elements:
        names.add("Jeff");
        names.add("Dan");
        names.add("Jeff");
        names.add("Joe");
        names.add("Annie");
        names.add("Anthony");
        names.add("Annie");

        Iterator nameIterator = names.iterator();
        while(nameIterator.hasNext()){
            System.out.println(nameIterator.next());
        }
    }
}
```

Output:

```
Annie
Anthony
Dan
Jeff
Joe
```

Example: a set from another collection

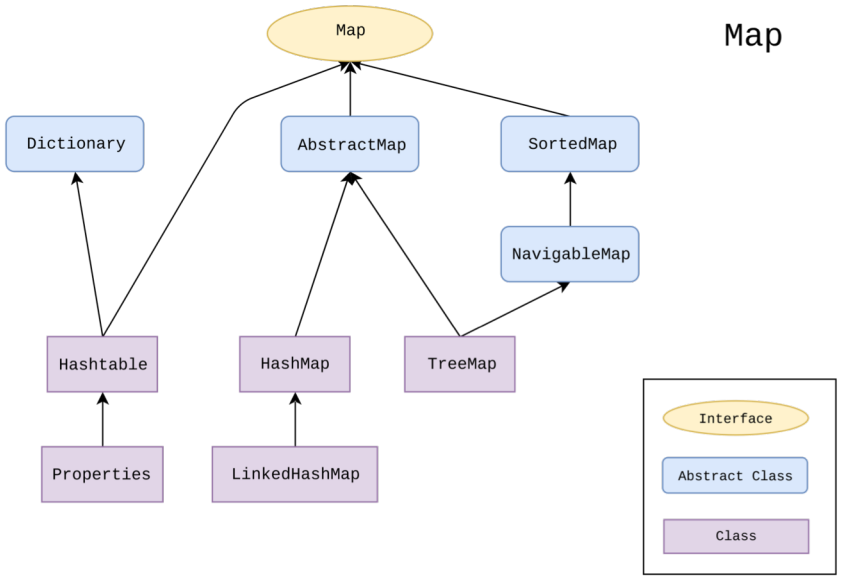
```java
public class Test {
    public static void main(String[] args){
        ArrayList<String> names = new ArrayList<String>();

        // add elements:
        names.add("Jeff");
        names.add("Dan");
        names.add("Jeff");
        names.add("Joe");
        names.add("Annie");
        names.add("Anthony");
        names.add("Annie");
        names.add("Tanya");
        names.add("Teddy");

        TreeSet<String> nameSet = new TreeSet<String>(names);

        Iterator nameIterator = nameSet.iterator();
        while(nameIterator.hasNext()){
            System.out.println(nameIterator.next());
        }
    }
}
```

Output:

```
Annie
Anthony
Dan
Jeff
Joe
Tanya
Teddy
```

Map interface

- Provides a framework for data structures that implement key-value pairs as elements
  - Dictionary-like data structure
- Each key is unique
- Each key maps to exactly one value



https://en.wikipedia.org/wiki/Java_collections_framework

Differences between classes representing map data structure

- HashMap
  - Hash table based implementation
  - Permits one null key and multiple null values
  - No guarantee of the order of key-value pairs
- LinkedHashMap
  - Linked list base implementation
  - Permits one null key and multiple null values
  - Maintains insertion order
- TreeMap
  - Red-black tree based implementation
  - Sorted by natural or Comparator order of the keys

Example: dictionary of names with HashMap

```java
public class Test {
    public static void main(String[] args){
        Map <Integer, String> names = new HashMap();

        // add elements:
        names.put(1,"Jeff");
        names.put(2,"Dan");
        names.put(3,"Teddy");
        names.put(4,"Joe");
        names.put(5,"Annie");
        names.put(6,"Anthony");

        // test if the map contains elements:
        System.out.println("Map contains value Joe: "+names.containsValue("Joe"));
        System.out.println("Map contains key 3: "+names.containsKey(3));
        System.out.println("Value at key=2: "+names.get(2));
        System.out.println("Map size: "+names.size());
        System.out.println("All values in the map: "+names.values());

        // traverse the map:
        System.out.println();
        Set dict = names.entrySet();
        Iterator nameIterator = dict.iterator();
        while(nameIterator.hasNext()){
            Map.Entry nameEntry = (Map.Entry)nameIterator.next();
            System.out.println(nameEntry.getKey()+": "+nameEntry.getValue());
        }

        // remove items from the map:
        names.remove(2);
        names.remove(5,"Annie");
    }
}
```

Output:

```
Map contains value Joe: true
Map contains key 3: true
Value at key=2: Dan
Map size: 6
All values in the map: [Jeff, Dan, Teddy, Joe, Annie, Anthony]

1: Jeff
2: Dan
3: Teddy
4: Joe
5: Annie
6: Anthony
```

Some of the ways to traverse a map

```java
public class Test {
    public static void main(String[] args){
        Map<String,Integer> names = new HashMap();

        // add elements:
        names.put("a",100);
        names.put("b",5);
        names.put("c",78);
        names.put("d",34);
        names.put("e",99);
        names.put("f",11);

        // Iterator with enterySet():
        Iterator<Map.Entry<String, Integer>> nameIterator = names.entrySet().iterator();
        while(nameIterator.hasNext()) {
            Map.Entry<String, Integer> nameEntry = nameIterator.next();
            System.out.println(nameEntry.getKey()+": "+nameEntry.getValue());
        }
        System.out.println();

        // Iterator with keySet():
        Iterator<String> keyIterator = names.keySet().iterator();
        while(keyIterator.hasNext()) {
            String k = keyIterator.next();
            System.out.println(k+": "+names.get(k));
        }
        System.out.println();

        // for-each loop with Map.Entry:
        for (Map.Entry<String, Integer> mapEntry : names.entrySet()) {
            System.out.println(mapEntry.getKey()+": "+mapEntry.getValue());
        }
        System.out.println();

        // for-each loop with keySet():
        for (String key : names.keySet()) {
            System.out.println(key+": "+names.get(key));
        }
        System.out.println();

        // forEach():
        names.forEach((k, v) -> System.out.println(k+": "+v));
    }
}
```

Sorting a map data structure

- Can sort by both keys and values
- Custom sort order can be achieved by using custom Comparator
- Sorting by key
  - Converting the map to TreeMap data structure is the easiest way to sort the map
  - We can sort the keys separately and then process the map entries based on that sorted order
    - E.g. *Collections.sort(arrayListOfKeys)*

Example: sort a map of names by key using TreeMap data structure

```java
public class Test {
    public static void main(String[] args){
        LinkedHashMap <Integer, String> names = new LinkedHashMap();

        // add elements:
        names.put(6,"Jeff");
        names.put(1,"Dan");
        names.put(3,"Teddy");
        names.put(2,"Joe");
        names.put(5,"Annie");
        names.put(4,"Anthony");

        // sort names:
        TreeMap<Integer, String> sortedNames = new TreeMap();
        sortedNames.putAll(names);

        // traverse the sorted map:
        for (Integer key : sortedNames.keySet()) {
            System.out.println(key+": "+sortedNames.get(key));
        }
    }
}
```

Output:

```
1: Dan
2: Joe
3: Teddy
4: Anthony
5: Annie
6: Jeff
```

Make a *TreeMap* from an already populated *LinkedHashMap*

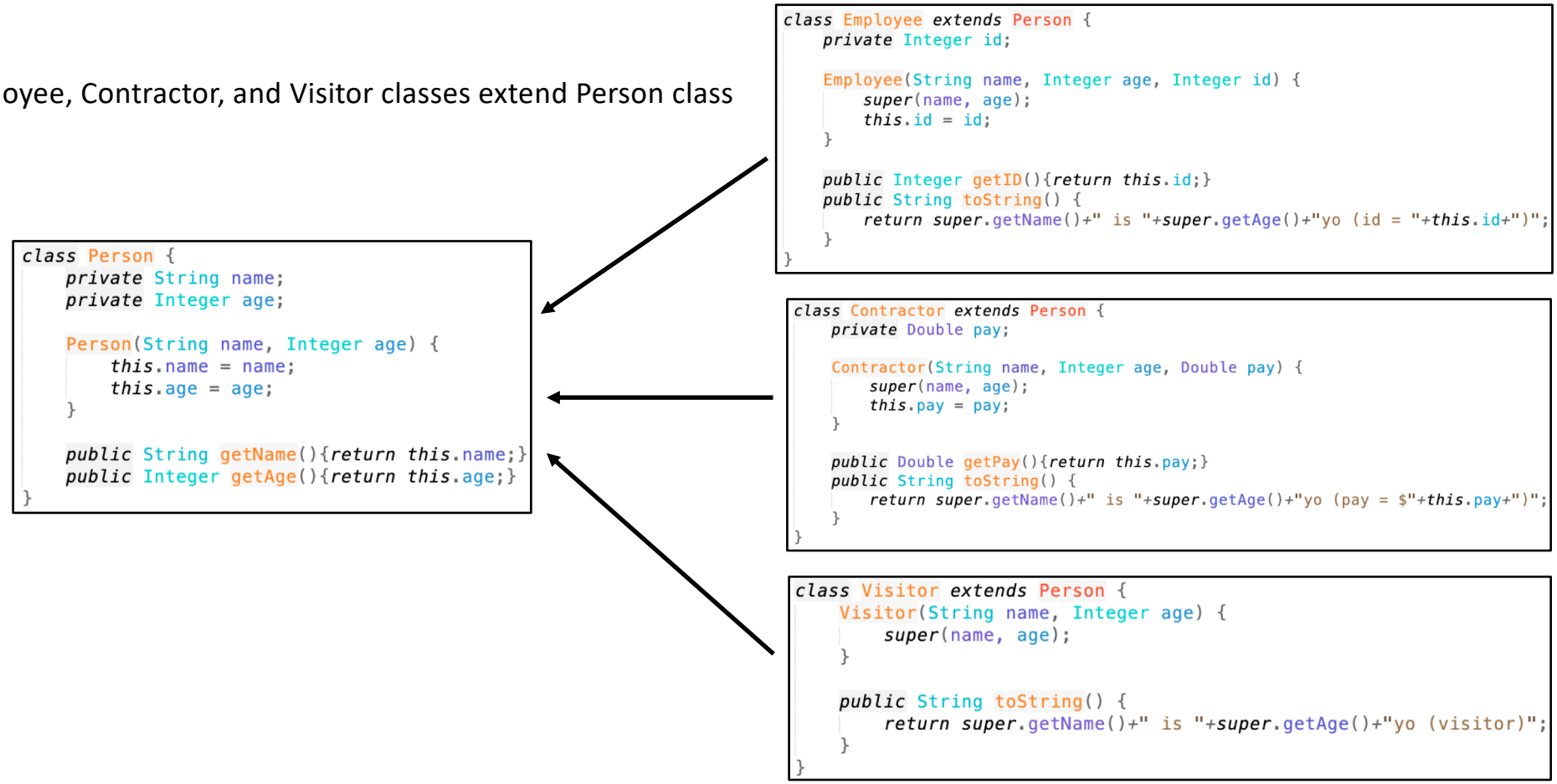
Example: alternatively, we can use TreeMap constructor to instantiate a map from another map

```
public class Test {
    public static void main(String[] args){
        LinkedHashMap <Integer, String> names = new LinkedHashMap();

        // add elements:
        names.put(6,"Jeff");
        names.put(1,"Dan");
        names.put(3,"Teddy");
        names.put(2,"Joe");
        names.put(5,"Annie");
        names.put(4,"Anthony");

        // sort names:
        TreeMap<Integer, String> sortedNames = new TreeMap(names);

        // traverse the sorted map:
        for (Integer key : sortedNames.keySet()) {
            System.out.println(key+": "+sortedNames.get(key));
        }
    }
}
```


Same example as before but we pass names as an argument to TreeMap constructor

Example: sorting a map of Person objects

Employee, Contractor, and Visitor classes extend Person class



```
class Person {
    private String name;
    private Integer age;

    Person(String name, Integer age) {
        this.name = name;
        this.age = age;
    }

    public String getName(){return this.name;}
    public Integer getAge(){return this.age;}
}
```

```
class Employee extends Person {
    private Integer id;

    Employee(String name, Integer age, Integer id) {
        super(name, age);
        this.id = id;
    }

    public Integer getID(){return this.id;}
    public String toString() {
        return super.getName()+" is "+super.getAge()+"yo (id = "+this.id+")";
    }
}
```

```
class Contractor extends Person {
    private Double pay;

    Contractor(String name, Integer age, Double pay) {
        super(name, age);
        this.pay = pay;
    }

    public Double getPay(){return this.pay;}
    public String toString() {
        return super.getName()+" is "+super.getAge()+"yo (pay = $"+this.pay+")";
    }
}
```

```
class Visitor extends Person {
    Visitor(String name, Integer age) {
        super(name, age);
    }

    public String toString() {
        return super.getName()+" is "+super.getAge()+"yo (visitor)";
    }
}
```

Example: sorting a map of Person objects (cont'd)

Map that contains Person objects as entries

```
public class Test {
    public static void main(String[] args){
        LinkedHashMap <Integer, Person> names = new LinkedHashMap <Integer, Person>();

        // add elements:
        names.put(3, new Employee("Jeff", 24, 101));
        names.put(8, new Contractor("Dan", 32, 15.75));
        names.put(1, new Employee("Teddy", 28, 102));
        names.put(5, new Visitor("Joe", 40));
        names.put(2, new Contractor("Annie", 35, 55.0));
        names.put(7, new Contractor("Anthony", 40, 34.0));
        names.put(9, new Employee("Casey", 20, 103));
        names.put(4, new Visitor("Lauren", 24));

        // sort names:
        TreeMap<Integer, Person> sortedNames = new TreeMap(names);

        // traverse the sorted map:
        for (Integer key : sortedNames.keySet()) {
            System.out.println(key+": "+sortedNames.get(key).toString());
        }
    }
}
```

As before, we use TreeMap data structure to sort unsorted map by keys

Output:

```
1: Teddy is 28yo (id = 102)
2: Annie is 35yo (pay = $55.0)
3: Jeff is 24yo (id = 101)
4: Lauren is 24yo (visitor)
5: Joe is 40yo (visitor)
7: Anthony is 40yo (pay = $34.0)
8: Dan is 32yo (pay = $15.75)
9: Casey is 20yo (id = 103)
```

Example: use Person object as keys for the map, sort based on the person's age

person objects are keys

```java
public class Test {
    public static void main(String[] args){
        LinkedHashMap <Person, Integer> names = new LinkedHashMap<Person, Integer>();

        Comparator<Person> myPersonComparator = new Comparator<Person>() {
            @Override
            public int compare(Person p1, Person p2) {
                return p1.getAge() - p2.getAge();
            }
        };

        // add elements:
        names.put(new Employee("Jeff", 24, 101),3);
        names.put(new Contractor("Dan", 32, 15.75),8);
        names.put(new Employee("Teddy", 28, 102),1);
        names.put(new Visitor("Joe", 40),5);
        names.put(new Contractor("Annie", 35, 55.0),2);
        names.put(new Contractor("Anthony", 40, 34.0),7);
        names.put(new Employee("Casey", 20, 103),9);
        names.put(new Visitor("Lauren", 24),4);

        // sort names:
        TreeMap<Person, Integer> sortedNames = new TreeMap<Person, Integer>(myPersonComparator);
        sortedNames.putAll(names);

        // traverse the sorted map:
        for (Person key : sortedNames.keySet()) {
            System.out.println(key+": "+sortedNames.get(key));
        }
    }
}
```

custom Comparator that compares two Person objects

Output:

```
Casey is 20yo (id = 103): 9
Jeff is 24yo (id = 101): 4
Teddy is 28yo (id = 102): 1
Dan is 32yo (pay = $15.75): 8
Annie is 35yo (pay = $55.0): 2
Joe is 40yo (visitor): 7
```

Use custom comparator as a constructor argument for TreeMap object

Example: sort map by value

```java
public class Test {
    public static void main(String[] args){
        LinkedHashMap <Integer, Person> names = new LinkedHashMap<Integer, Person>();
        ArrayList<Person> personList = new ArrayList<Person>();
        LinkedHashMap <Integer, Person> sortedNames = new LinkedHashMap<Integer, Person>();

        Comparator<Person> myPersonComparator = new Comparator<Person>() {
            @Override
            public int compare(Person p1, Person p2) {
                return p1.getAge() - p2.getAge();
            }
        };

        // add elements:
        names.put(3, new Employee("Jeff", 24, 101));
        names.put(8, new Contractor("Dan", 32, 15.75));
        names.put(1, new Employee("Teddy", 28, 102));
        names.put(5, new Visitor("Joe", 40));
        names.put(2, new Contractor("Annie", 35, 55.0));
        names.put(7, new Contractor("Anthony", 40, 34.0));
        names.put(9, new Employee("Casey", 20, 103));
        names.put(4, new Visitor("Lauren", 24));

        // sort by value:
        for (Map.Entry<Integer, Person> person : names.entrySet()) {
            personList.add(person.getValue());
        }
        Collections.sort(personList, myPersonComparator);
        for(Person p : personList) {
            for(Map.Entry<Integer, Person> mapEntry : names.entrySet()) {
                if(mapEntry.getValue().equals(p)) {
                    sortedNames.put(mapEntry.getKey(), p);
                }
            }
        }

        // traverse the sorted map:
        for (Integer key : sortedNames.keySet()) {
            System.out.println(key+": "+sortedNames.get(key));
        }
    }
}
```

store map values in ArrayList

sort values

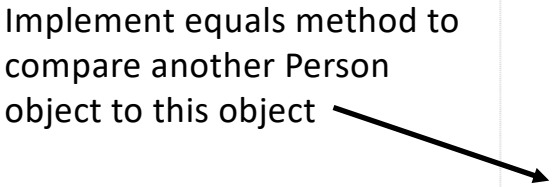
reconstruct the map with sorted value order

Output:

```
9: Casey is 20yo (id = 103)
3: Jeff is 24yo (id = 101)
4: Lauren is 24yo (visitor)
1: Teddy is 28yo (id = 102)
8: Dan is 32yo (pay = $15.75)
2: Annie is 35yo (pay = $55.0)
5: Joe is 40yo (visitor)
7: Anthony is 40yo (pay = $34.0)
```

Example: sort map by value (cont'd)

Implement equals method to compare another Person object to this object



```
class Person {
    private String name;
    private Integer age;

    Person(String name, Integer age) {
        this.name = name;
        this.age = age;
    }

    public String getName(){return this.name;}
    public Integer getAge(){return this.age;}
    public boolean equals(Person p) {
        if(p.getName() == this.name && p.getAge() == this.age){return true;}
        else{return false;}
    }
}
```

Modifiable vs. unmodifiable collections

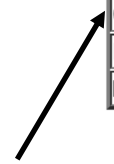
- Collections that do not allow modifications (add, remove, clear) are called “unmodifiable”
  - No guarantee that content will not change by indirect methods though
  - *copyOf()* collections method returns unmodifiable collection
    - E.g. *List.copyOf()*, *Set.copyOf()*, *Map.copyOf()*
- Immutable collections are guaranteed to not change contents

Bird’s eye view of collections available in Java

General-purpose Implementations

Interfaces	Hash table Implementations	Resizable array Implementations	Tree Implementations	Linked list Implementations	Hash table + Linked list Implementations
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue					
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

www.docs.oracle.com



PriorityQueue

Concluding remarks

- Read Java API for these data structures!
  - Get familiar with methods offered by each data structure
  - https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/util/package-summary.html
- Other data structures exist
  - We did not cover every single one