Chapter 1011: Java Collections Framework

WEEK 3: LISTS, SETS, AND MAPS

Collections

- A collection is an object that stores a group of other objects (its “elements”)
- An ArrayList is an example of a collection
- A collection uses a data structure internally to store its elements (e.g. the ArrayList uses an array)
- Collections are categorized by the types of elements they store, the operations they allow you to perform, and the speed or efficiency of those operations

Examples of Collections

- **List:** An ordered collection of elements, often accessed by integer indices or iteration
- **Stack:** A collection in which the last element added is the first one removed ("LIFO": last in, first out)
- **Queue:** A collection in which elements are removed in the same order they were added "FIFO": first in, first out)
- **Set:** A collection of elements guaranteed not to contain duplicates
- **Map:** A collection of key/value pairs, in which each (unique) key is associated with a corresponding value

Java Collections Framework

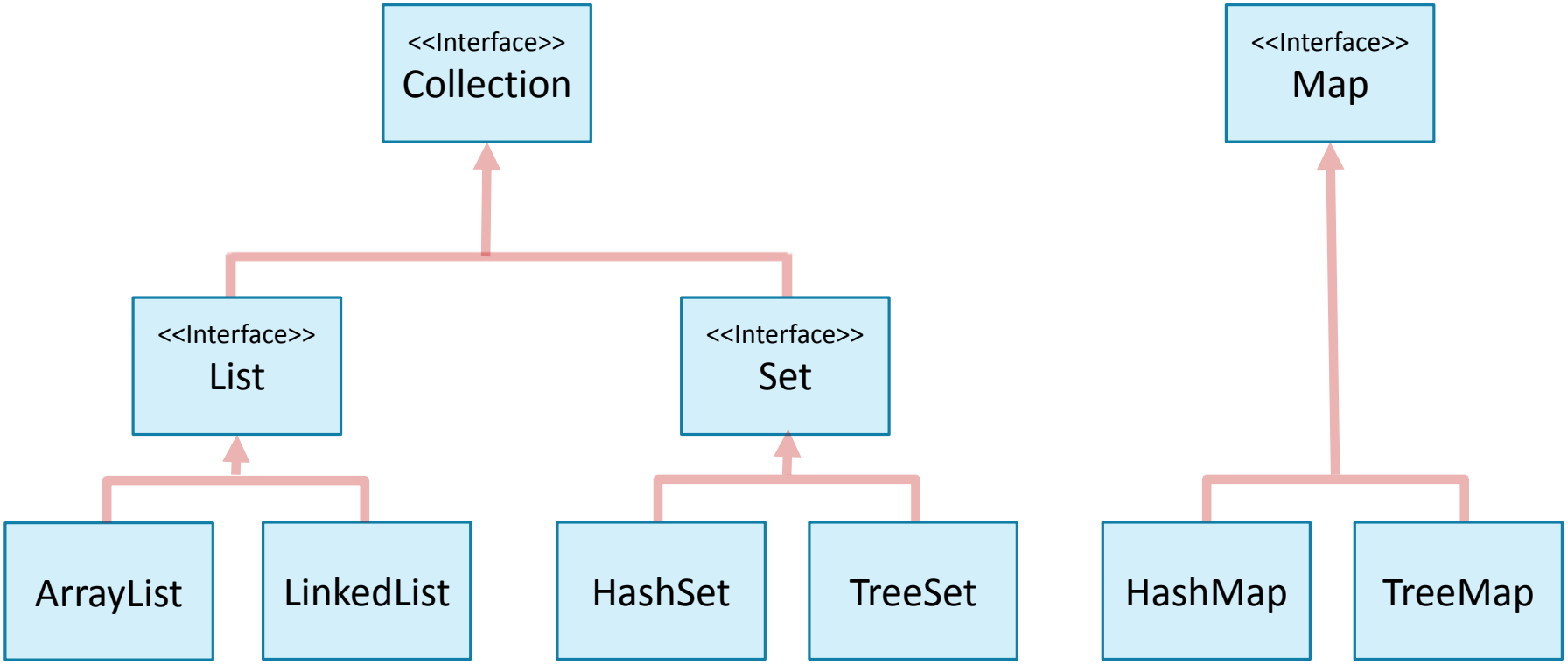
- Java provides a large group of useful collections
- We will look at just a subset of these
- More information available online: http://docs.oracle.com/javase/tutorial/collections/
- This unit will introduce lists, sets, and maps
- Stacks and queues will be covered later in the course
- All collections except Map implement the Collection interface

The Collection Interface

The Collection interface specifies a number of useful methods:

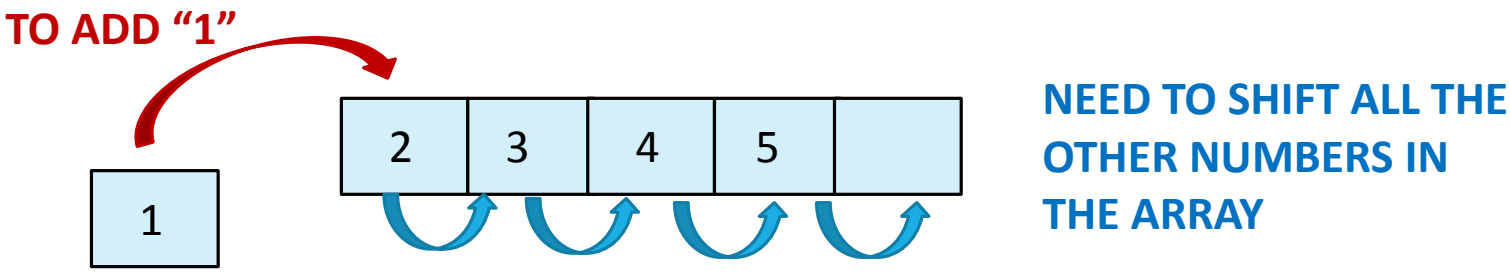
- add(element)
- addAll(collection)
- clear()
- remove(element)
- removeAll(collection)
- contains(element)
- isEmpty()
- size()

Java Collections Framework (abridged view)



ArrayList vs. LinkedList

- ArrayLists and LinkedLists both perform the same operations, but there are times when one is preferred over the other
- For example, adding and removing items to or from an ArrayList object is laborious
  - Since the ArrayList is implemented using an array, items generally need to be shifted for adding and removing:

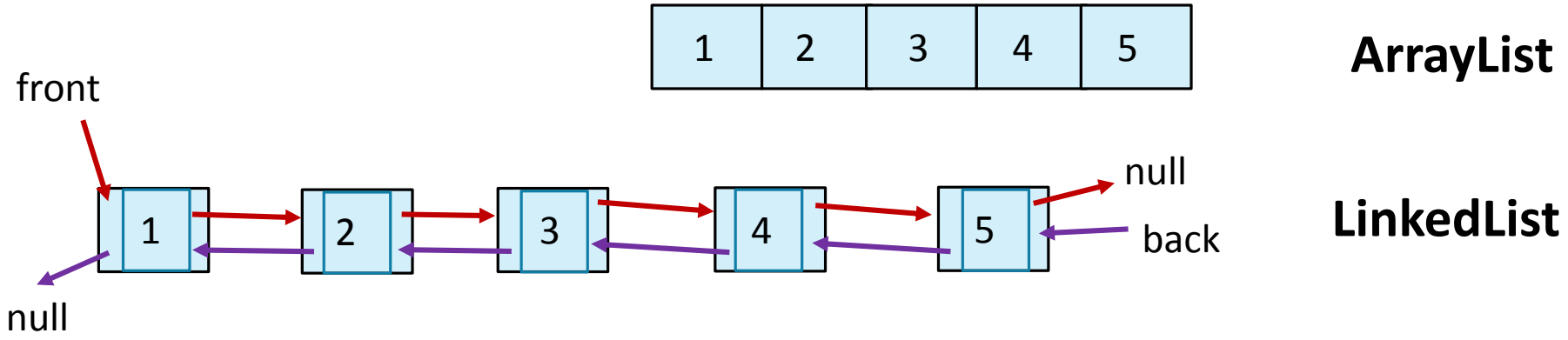


Adding & Removing List Items

- Removing items from an array (or an ArrayList) generally involves similar shifting of elements
- A LinkedList gives better performance in these cases, because it stores items very differently – using a data structure called a linked list
- In a linked list, elements are stored in individual nodes, along with references to other nodes (at a minimum, to the next node in the list, and often to both the next node and the previous node)

Basic linked list structure

- Will be covered in more depth in Chapter 16
- The linked list object keeps references to the front and back nodes:
- Each node stores data along with references to the node preceding it and the next node after it



Adding and removing elements can be achieved by changing references

Iterators

- Unlike arrays, linked lists do NOT provide fast random access
- In order to reach an element in a linked list, you need to "walk through" the list from either the front or the back reference to the correct position
- Each node is accessed via its "next" or "previous" reference
- This is a slow, inefficient process – but a special object called an iterator provides an efficient way to examine all the elements in a linked list in sequential order

Iterators, continued…

- Every collection provides iterators to access its elements
- To get an iterator from most collections, call the iterator method:

```
Iterator<E> itr = list.iterator;
while (itr.hasNext()){
        <do something with itr.next()>;}
```

- before trying to access itr.next(), you generally want to test that there are still elements to access: use itr.hasNext()

NOTE: itr.next() returns the next element in the list. Operations performed on this returned value *do not change* the value stored in the list

Removing items from a list

- Items can be removed from a LinkedList object using the class's remove method
- But, as mentioned earlier, this can be inefficient
- A nice feature of iterators is that they have remove methods of their own that allow list items to be removed:

  itr.remove() will remove the element most recently removed by the iterator

- However, there is no iterator add method: attempting to add elements to a list while iterating over it will result in a *ConcurrentModificationException*

ListIterator

- There is a more advanced version of Iterator, the **ListIterator**, that does allow you to add elements, replace elements, and reverse iteration (from back to front)
- The ListIterator is not a class we will cover further, but if you want to experiment with it, feel free to use it in any of your programs that would need iterators (unless specifically asked not to)
- http://docs.oracle.com/javase/7/docs/api/java/util/ListIterator.html

Caution concerning iterators

- Be careful not to call itr.next() too often!
- If you want to perform several operations on a list element, it is probably best to store the value of the element in a variable:

```
System.out.println(fruitSalad);

Iterator<String> itr = fruitSalad.iterator();
while(itr.hasNext()){
    itr.next().toUpperCase();
    System.out.println(itr.next() + "!");
}
```

Problems @ Javadoc Declaration Console

```
<terminated> BookCode [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe
[orange, banana, apple, strawberry]
banana!
strawberry!
```

This code skips items and fails to capitalize anything that is printed

```
System.out.println(fruitSalad);

Iterator<String> itr = fruitSalad.iterator();
while(itr.hasNext()){
    String current = itr.next().toUpperCase();
    System.out.println(current + "!");
}
```

Problems @ Javadoc Declaration Console

```
<terminated> BookCode [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe
[orange, banana, apple, strawberry]
ORANGE!
BANANA!
APPLE!
STRAWBERRY!
```

This code works as intended

ArrayList vs. LinkedList (revisited)

Collection	Strengths
ArrayList	• Random access: any element can be accessed quickly • Adding and removing at the end of the list is fast
LinkedList	• Adding and removing at either the beginning or end of the list is fast • Removing during a sequential access with an iterator is fast (adding with a special ListIterator is also fast) • Unlike with arrays, there is no need to expand when full • Can be more easily used as a queue than arrays can (more on this later)

Abstract Data Types (ADTs)

- An ADT is the specification of a type of data and the operations that can be used with that data type
- An ADT does NOT specify **HOW** the operations should be implements
- In Java, ADTs are specified by ***interfaces***
  - The ADT's operations are specified by the ***methods*** of its interface

ADTs: Advanced Topic

- Most likely, you won't need this information for any programs you write for this class – but it is good to know…
- It is considered good practice to declare variables and parameters of a collection type using the appropriate ***interface*** type for that ADT, rather than the active class's type:

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

rather than:

```
LinkedList<Integer> list = new LinkedList<>():
```

Advanced Topic, continued…

- This gives you flexibility to change the implementation at a later point in time
- If you write all your methods declaring List parameters, return types, and fields, rather than LinkedList parameters, return types, and fields, you can later change the implementation to ArrayList:

List<Integer> list = new ArrayList<>():

and everything should work fine, with no additional changes to your code

Advanced Topic, conclusion.

- You can generalize even further and declare parameters and variables as collection types:
- However, you lose direct access to the ArrayList class methods

```
Collection<Integer> myList = new ArrayList<>();
```

- To access any of them (e.g. the get method), you would need to use casting to create an actual ArrayList object from your Collection object

```
ArrayList<Integer> myArrayList = (ArrayList)myList;
myArrayList.get(2);
```

Collections class

NOTE: the Collections class IS NOT THE SAME as the Collection interface

- The Collections class contains useful static methods that operate on lists, including (see p. 711 in your text):
  - binarySearch(list, value) – search a list for a specific value
  - copy(destList, sourceList) – copy elements of a list to a second list
  - max(list) & min(list) – return element with the highest/lowest value
  - reverse(list) – reverse order of list elements
  - swap(list, index1, index2) – switch the elements at the given indices

Sets

- Searching any kind of list is **time consuming**, as it generally requires sequentially checking each individual element in the list
- Another issue with lists is that there is **no easy way** to prevent the occurrence of **duplicate** elements (would need to check each item in the list before any new element was added)
- A different abstract data type, the ***set***, is better suited to both these tasks
- A set is a collection that **CANNOT** contain duplicates

Java set implementations

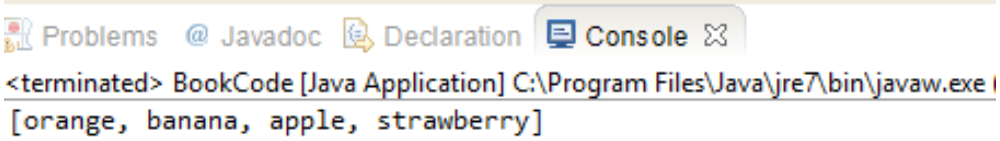
- The two main implementations of the Java Collections Framework's Set interface are **HashSet** and **TreeSet**
- **HashSet** uses a special internal array called a ***hash table*** that places elements into specific positions based on integers called ***hash codes*** (more on this in Chapter 18)
- The drawback of HashSet is that items are not stored in a predictable order
- **TreeSet** stores elements in an internal linked data structure called a ***binary search tree*** (more in Chapter 17) and thus does store elements in order
- The drawback of TreeSet is that is requires its data to be of a type that implements the ***Comparable interface***

HashSet vs. TreeSet

Collection	Description & Strengths
HashSet	• Stores elements in a special array called a hash table • Extremely fast performance for add, contains, and remove tasks • Can be used with any type of objects as its elements
TreeSet	• Stores elements in a linked data structure called a binary search tree • Elements are stored in sorted order • Must be used with elements that can be compared (such as Integer or String)

HashSet examples

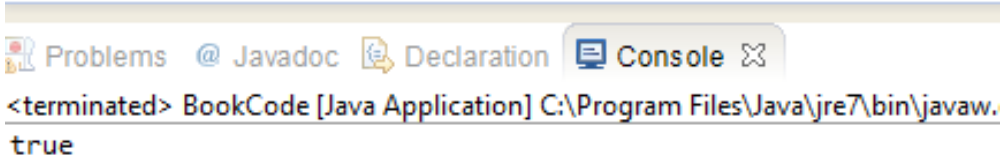
```
HashSet<String> fruitSalad = new HashSet<>();
fruitSalad.add("apple");
fruitSalad.add("banana");
fruitSalad.add("apple");
fruitSalad.add("orange");
fruitSalad.add("orange");
fruitSalad.add("strawberry");

System.out.println(fruitSalad);
```



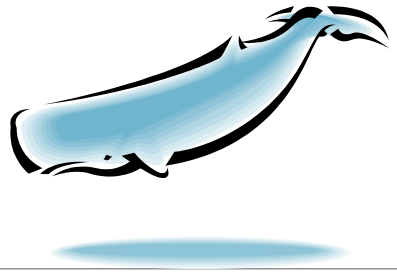
Although 6 **add** operations are listed in the code, the set will only contain the 4 unique elements

Use the **contains** method to search a set for a given item – such searches are performed incredibly quickly even on very large sets

```
boolean hasApple = fruitSalad.contains("apple");
System.out.println(hasApple);
```



Another set example



Sets can be used to return unique collections of elements – the following code generates the collection of unique words in the novel *Moby Dick*

```
HashSet<String> words = new HashSet<>();
Scanner in = new Scanner(new File("mobydick.txt"));
while(in.hasNext()){
    String word = in.next();
    word = word.toLowerCase();
    words.add(word);
}
System.out.println("Number of unique words = " + words.size());
```

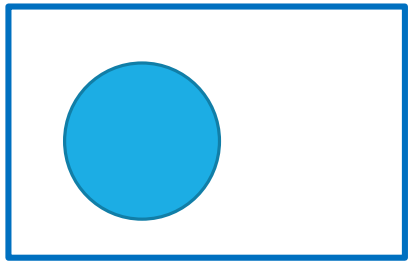
```
Number of unique words = 30368
```

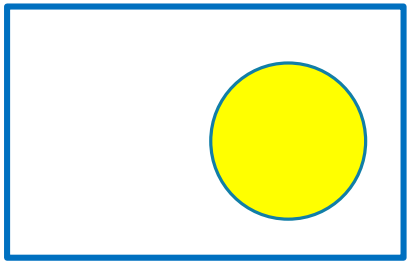
Now you know…

If you wanted to iterate over the words in the set “words”, you would again use an iterator.

Set operations - Diagrams

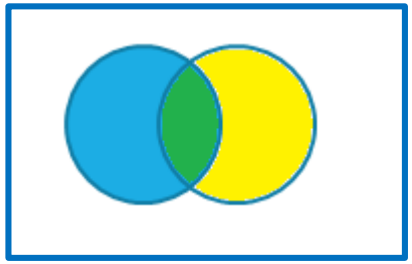
Set A

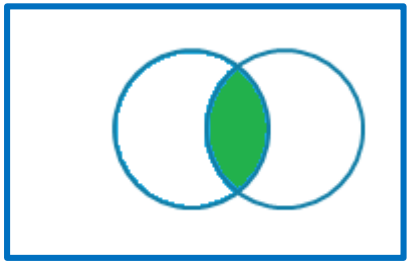




Set B

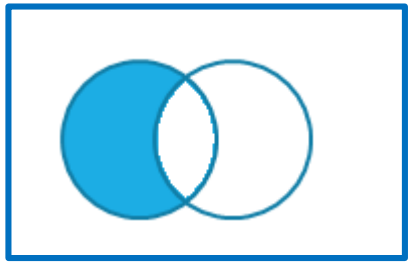
A U B
Union
addAll()





A ∩ B
Intersection
retainAll()

A – B
Difference
removeAll()



One of these may be useful for Weekly Problems – Week 3
More details on pp. 720-721

Set operations (in words)

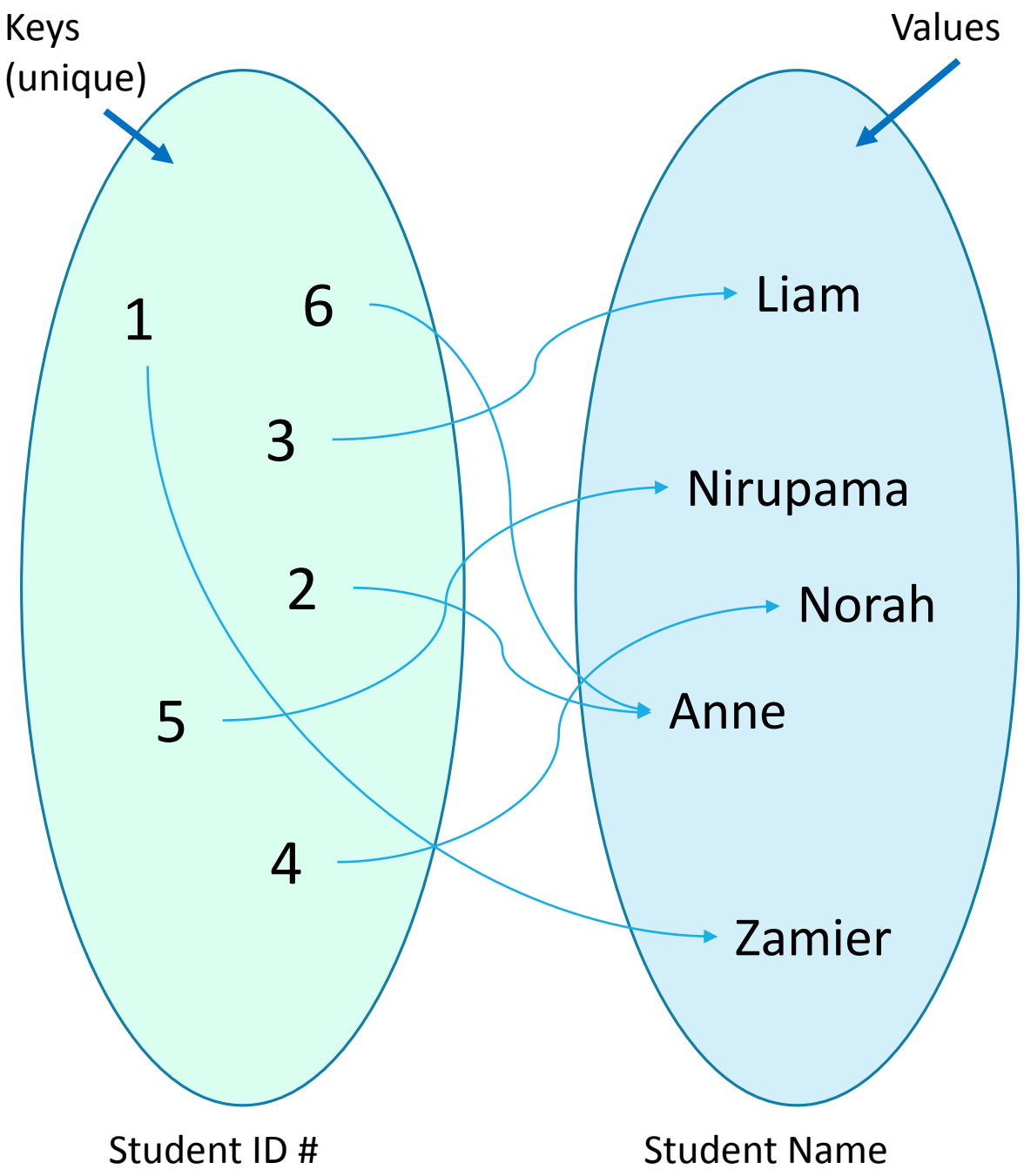
Method (called on SetA)	Description
addAll(setB)	Performs *union*: SetA now contains **ALL** elements that were in SetA, SetB or both
retainAll(setB)	Performs *intersection*: SetA now contains **ONLY** the elements that were in **BOTH** SetA **AND** SetB
removeAll (setB)	Performs *difference*: SetA now contains **ONLY** the elements that were **NOT** in SetB (i.e. not held in common with SetB)
containsAll(setB)	Returns true if SetA is a superset of SetB (i.e. if **ALL** elements of SetB are **ALSO** elements of SetA)

Maps

- In many data processing tasks, it is useful to link pairs of objects (e.g. student ID #s and student names)
- A **map** is a collection that associates (unique) objects called keys with objects called values
- In the HashSet example finding unique words in *Moby Dick*, perhaps we would have also liked to keep track of how many times each word occurred. A map could associate word counts (*values*) with **unique** words (*keys*).

Maps

Maps associate unique keys to values

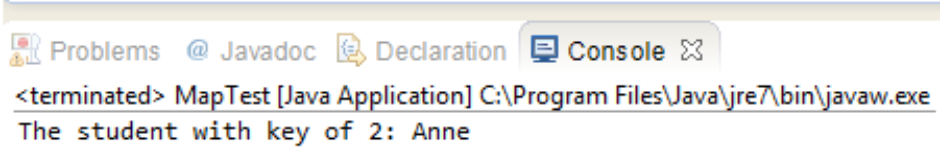


Useful map methods

Method	Description
clear()	Removes all keys and values from the map
containsKey(key)	Returns true if the given key maps to some value in this map
containsValue(value)	Returns true if some key maps to the given value in this map
get(key)	Returns the value associated with this key, or null if not found
isEmpty()	Returns true if this collection contains no keys or values
keySet()	Returns a Set of all the keys in this map
put(key, value)	Associates the given key with the given value
putAll(map)	Adds all key/value mappings from the given map to this map
remove(key)	Removes the given key and its associated value from this map
size()	Returns the number of key/value mappings in this map
values()	Returns a Collection of all the values in this map

Creating a map

```
public static void main(String[] args) {
    Map<Integer, String> studentIDMap = new HashMap<>();
    studentIDMap.put(1, "Zamier");
    studentIDMap.put(2, "Anne");
    studentIDMap.put(3, "Liam");
    studentIDMap.put(4, "Norah");
    studentIDMap.put(5, "Nirupama");
    studentIDMap.put(6,  "Anne");

    String currentStudent = studentIDMap.get(2);
    System.out.println("The student with key of 2: " + currentStudent);
}
```



Add a key/value pair to the map using the map's put method

Notice that you need to specify types for **BOTH** the key & the value: **Map<Integer, String>**

Unlike most collections, a map ***doesn't*** have an **iterator** method. Instead, it has a pair of methods: **keySet** and **values**. These methods return a Set of keys and a Collection of values, respectively.

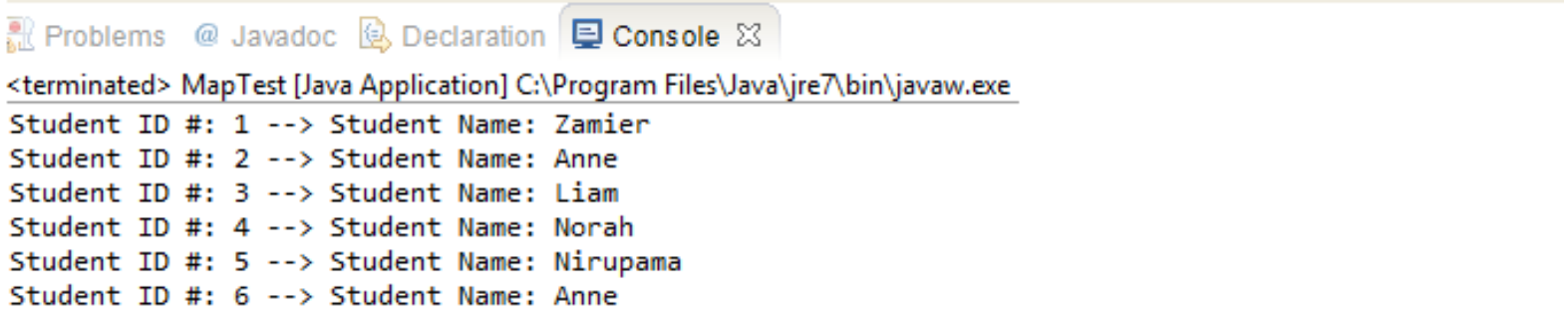
Map Views (keySet & values)

- Since keys must be unique, a set is an appropriate collection in which to store keys
- Values, however, don't need to be unique
- One you have the set of keys returned by keySet, you can loop over the keys. If you also want the values, you can look up the value associated with each key:

```
for(int StudentID : studentIDMap.keySet()){
        String name = studentIDMap.get(StudentID);
        System.out.println(StudentID + ": " + name);
}
```

Sample code & output

```
        for(int studentID: studentIDMap.keySet()){
            String name = studentIDMap.get(studentID);
            System.out.println("Student ID #: " + studentID + " --> Student Name: " + name);
        }
```



Can map from key to value easily, but CANNOT easily map back from a value to its key.

Comparison of Lists, Sets, & Maps

ADT	Implementations	Description	Weaknesses	Example Usages
List	ArrayList, LinkedList	A sequence of elements arranged in order of insertion	Slow to search, slow to add/remove arbitrary elements	List of accounts; prime numbers; the lines of a file
Set	HashSet, TreeSet	A set of unique elements that can be searched quickly	Does not have indices; user cannot retrieve arbitrary elements	Unique words in a book; lottery ticket numbers
Map	HashMap, TreeMap	A group of associations between pairs of “key” and “value” objects	Not a good general-purpose collection; cannot easily map backward from a value to its key	Word counting; phone book creation