Tutorial #11

SFWR ENG / COMP SCI 2S03

Interfaces and Java Collections

Dillon Dixon

Week of November 18, 2013

Interfaces

- Back in tutorial #6 we talked about *interfaces*.

- We can recall: *An interface defines a specification or contract that a class must meet to be defined as an instance of that interface.*

Interfaces

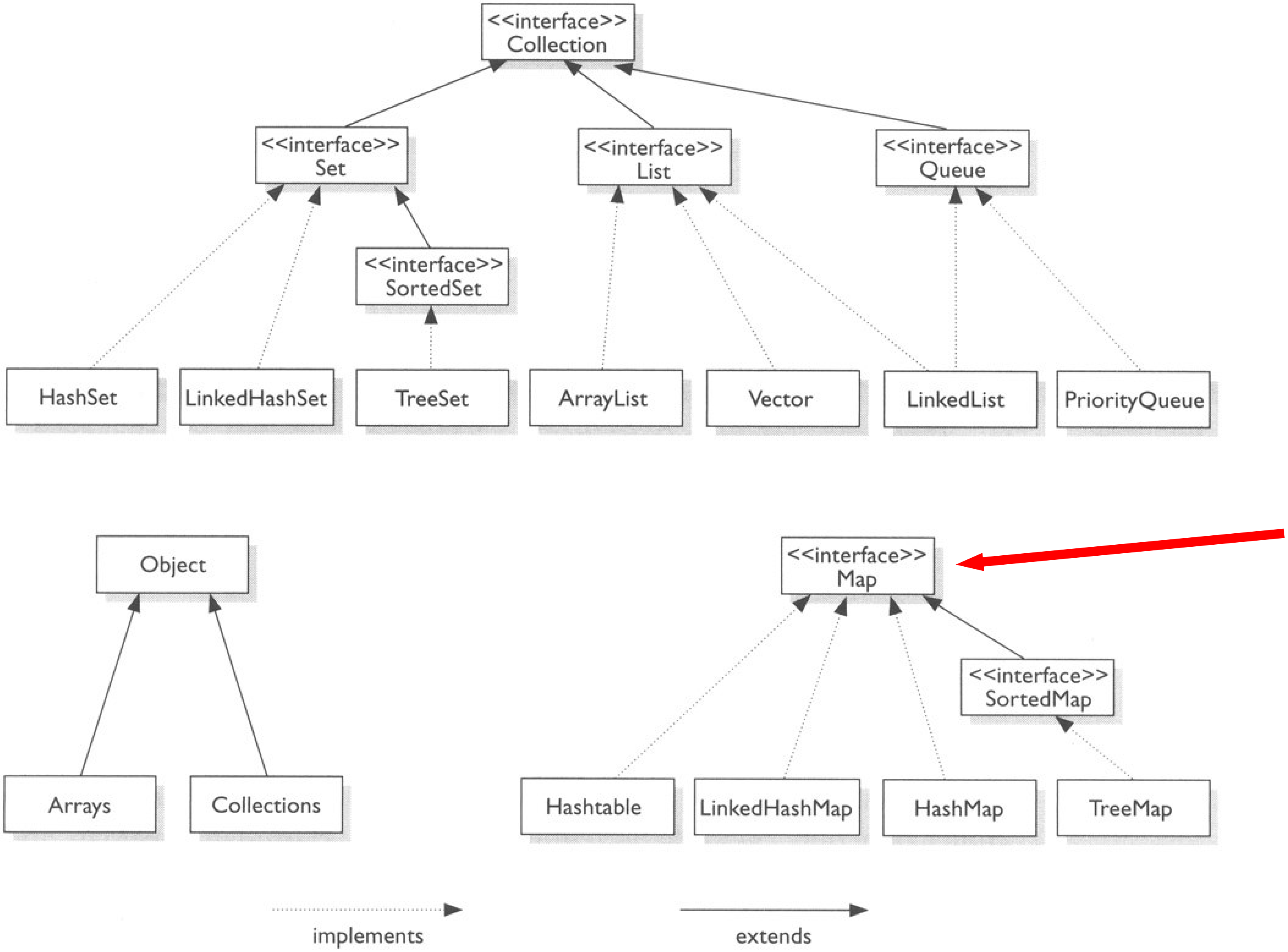
- Aside from designing our own interfaces, the Java framework has many of its own built in.
- One of the most popular set of interfaces are those found under the *Java Collections* framework.

Collections

- Java *collections* are a set of interfaces and classes used for storing “collections” of information.

- A *collection* is just a grouping of information, like an array, but with more flexibility.

What is in “collections”?



Not a true “collection”, but still worth mentioning!

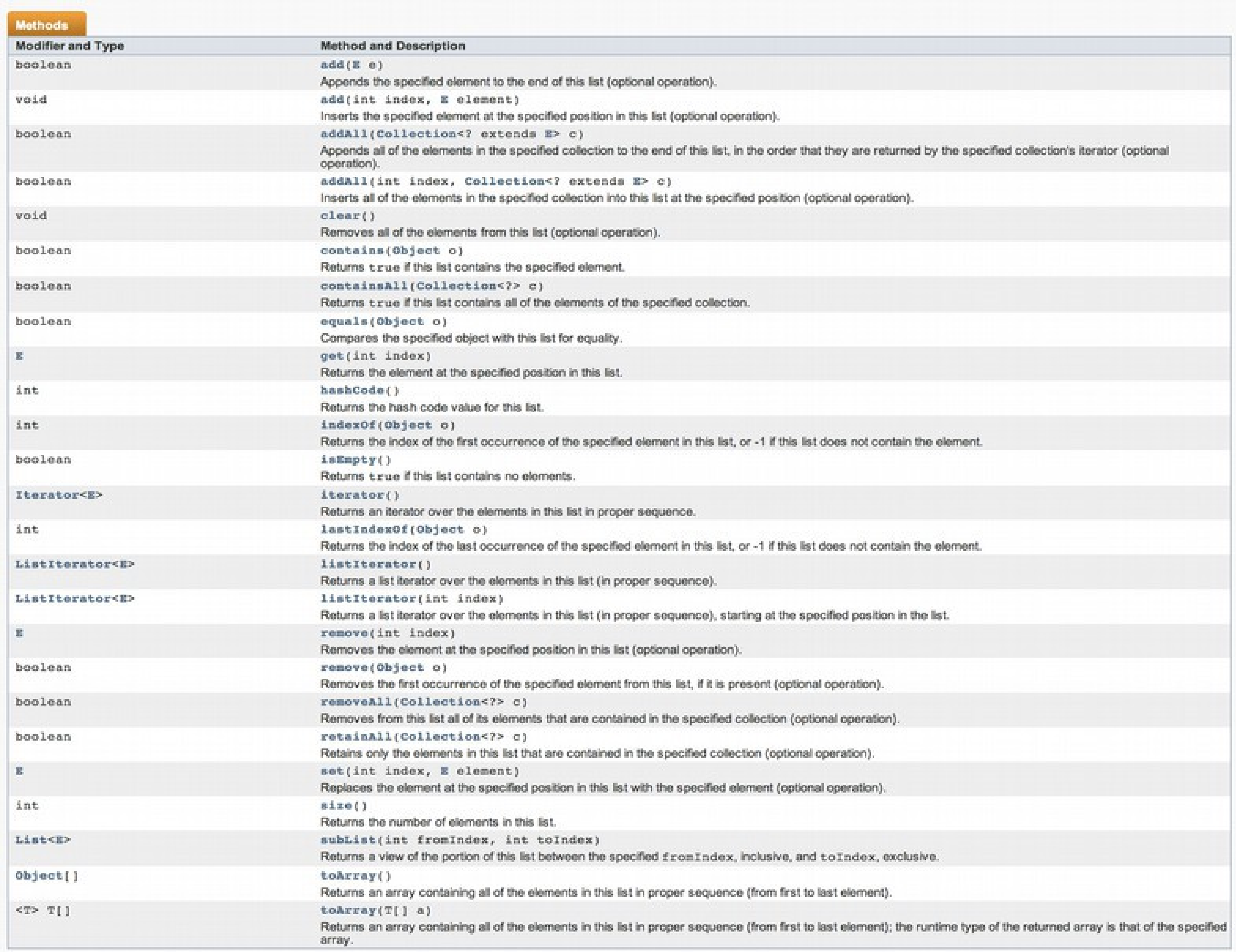
Interesting Interfaces

- The main interfaces we will look at today are:
  - List
  - Set
  - Map

List

- Found under *java.util.List* package.
- Represents a list of data, <u>ordered</u> sequentially, like in an array.
- Supports adding, removing, searching and inserting, among other operations.

The List Interface



Methods

Modifier and Type	Method and Description
boolean	add(E e) Appends the specified element to the end of this list (optional operation).
void	add(int index, E element) Inserts the specified element at the specified position in this list (optional operation).
boolean	addAll(Collection<? extends E> c) Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator (optional operation).
boolean	addAll(int index, Collection<? extends E> c) Inserts all of the elements in the specified collection into this list at the specified position (optional operation).
void	clear() Removes all of the elements from this list (optional operation).
boolean	contains(Object o) Returns true if this list contains the specified element.
boolean	containsAll(Collection<?> c) Returns true if this list contains all of the elements of the specified collection.
boolean	equals(Object o) Compares the specified object with this list for equality.
E	get(int index) Returns the element at the specified position in this list.
int	hashCode() Returns the hash code value for this list.
int	indexOf(Object o) Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
boolean	isEmpty() Returns true if this list contains no elements.
Iterator<E>	iterator() Returns an iterator over the elements in this list in proper sequence.
int	lastIndexOf(Object o) Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
ListIterator<E>	listIterator() Returns a list iterator over the elements in this list (in proper sequence).
ListIterator<E>	listIterator(int index) Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list.
E	remove(int index) Removes the element at the specified position in this list (optional operation).
boolean	remove(Object o) Removes the first occurrence of the specified element from this list, if it is present (optional operation).
boolean	removeAll(Collection<?> c) Removes from this list all of its elements that are contained in the specified collection (optional operation).
boolean	retainAll(Collection<?> c) Retains only the elements in this list that are contained in the specified collection (optional operation).
E	set(int index, E element) Replaces the element at the specified position in this list with the specified element (optional operation).
int	size() Returns the number of elements in this list.
List<E>	subList(int fromIndex, int toIndex) Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.
Object[]	toArray() Returns an array containing all of the elements in this list in proper sequence (from first to last element).
<T> T[]	toArray(T[] a) Returns an array containing all of the elements in this list in proper sequence (from first to last element); the runtime type of the returned array is that of the specified array.

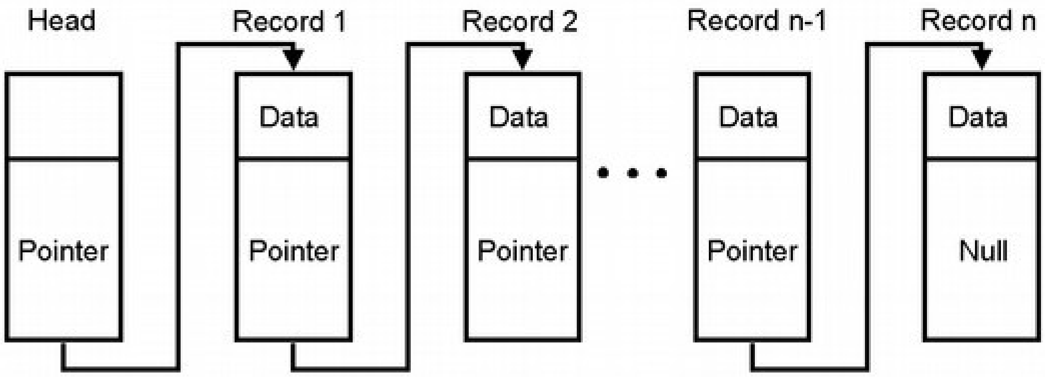
List

- There are many different *List* implementations for specific purposes (i.e. thread safety), but these two are typically the most popular.

  - LinkedList
  - ArrayList

LinkedList

- *LinkedList* was introduced in tutorial 10. Each element is a node in a “chain” of elements.



ArrayList

- *ArrayList* is simpler, and uses an array to store elements.

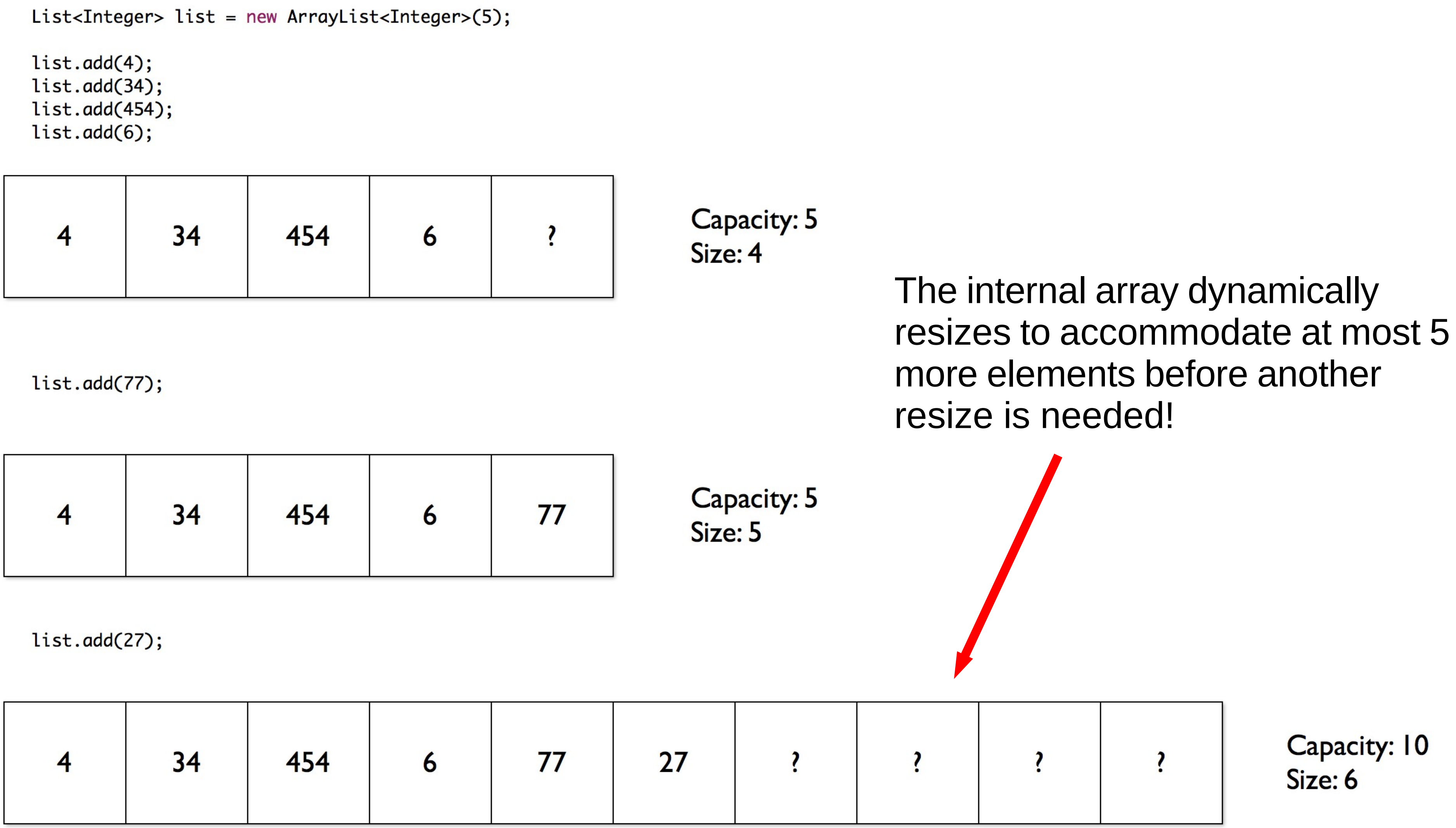
- It is much faster for *random* searching than *LinkedList* is, but can be much more costly to insert and delete from.

ArrayList

- *ArrayList* is typically optimized with a *capacity* to reduce the amount of resizing and copying.

```
List<Integer> list = new ArrayList<Integer>(5);
```

ArrayList - Internals



```
List<Integer> list = new ArrayList<Integer>(5);

list.add(4);
list.add(34);
list.add(454);
list.add(6);
```

4	34	454	6	?

Capacity: 5
Size: 4

```
list.add(77);
```

4	34	454	6	77

Capacity: 5
Size: 5

The internal array dynamically resizes to accommodate at most 5 more elements before another resize is needed!

```
list.add(27);
```

4	34	454	6	77	27	?	?	?	?

Capacity: 10
Size: 6

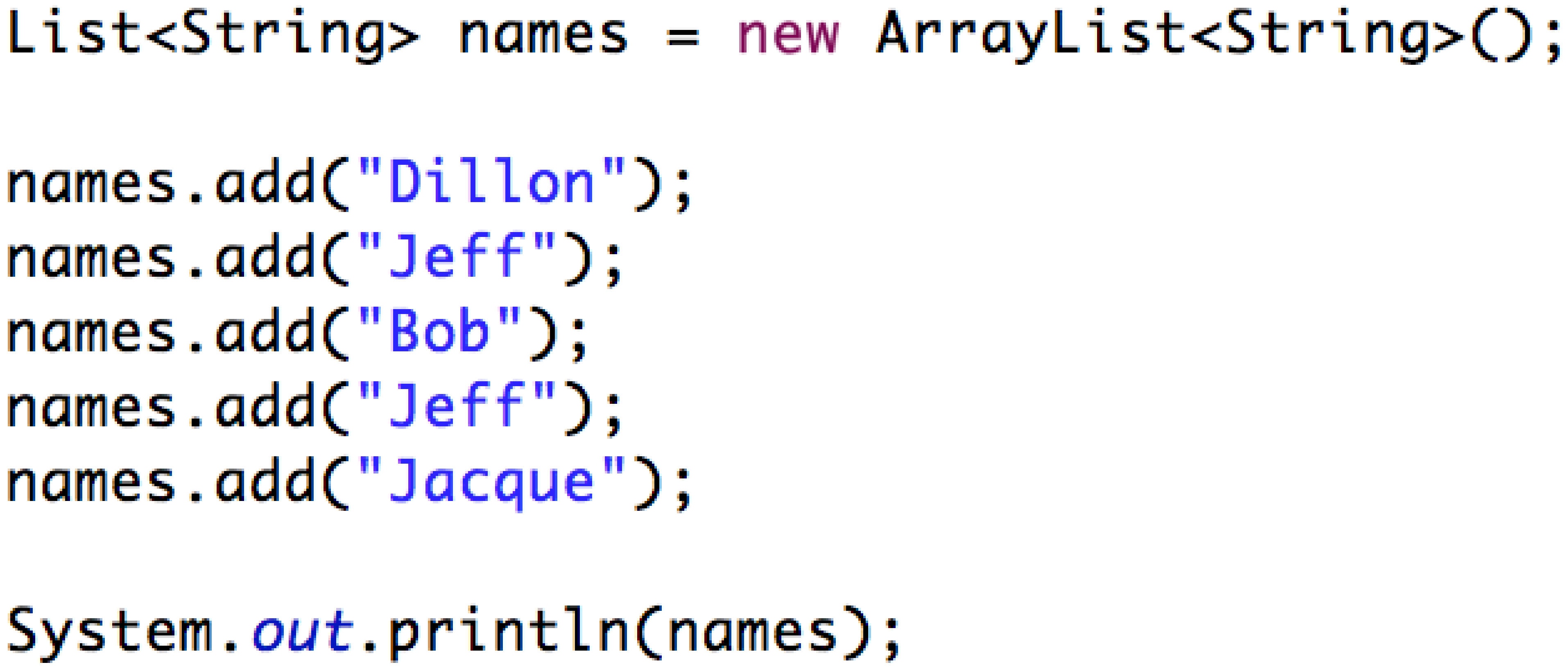
Which do I use?

- If you frequently insert/remove data and typically only access data sequentially, use *LinkedList*.

- If you do lots of random data access, and do not typically do deletions and inserts aside from the end of the list, use *ArrayList.*

- When in doubt – choose *ArrayList*!

List - Usage



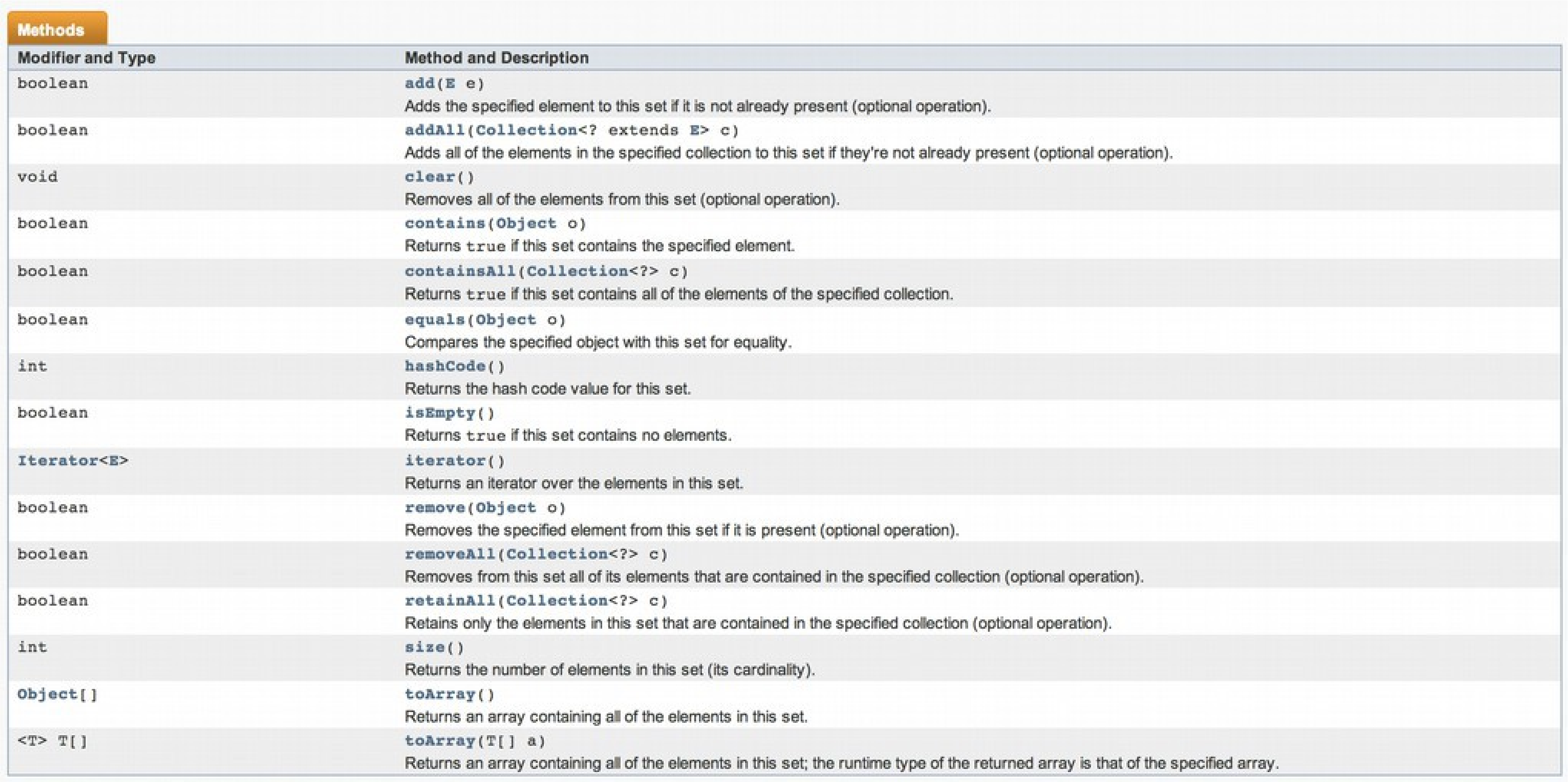
```
List<String> names = new ArrayList<String>();

names.add("Dillon");
names.add("Jeff");
names.add("Bob");
names.add("Jeff");
names.add("Jacque");

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

Prints: “[Dillon, Jeff, Bob, Jeff, Jacque]”

Set

- Found under the *java.util.Set* package.
- Represents a *set* of data.
- A *set* is an unordered group of *unique* elements.
- Supports adding, removing, searching and inserting, among other operations.

The Set Interface



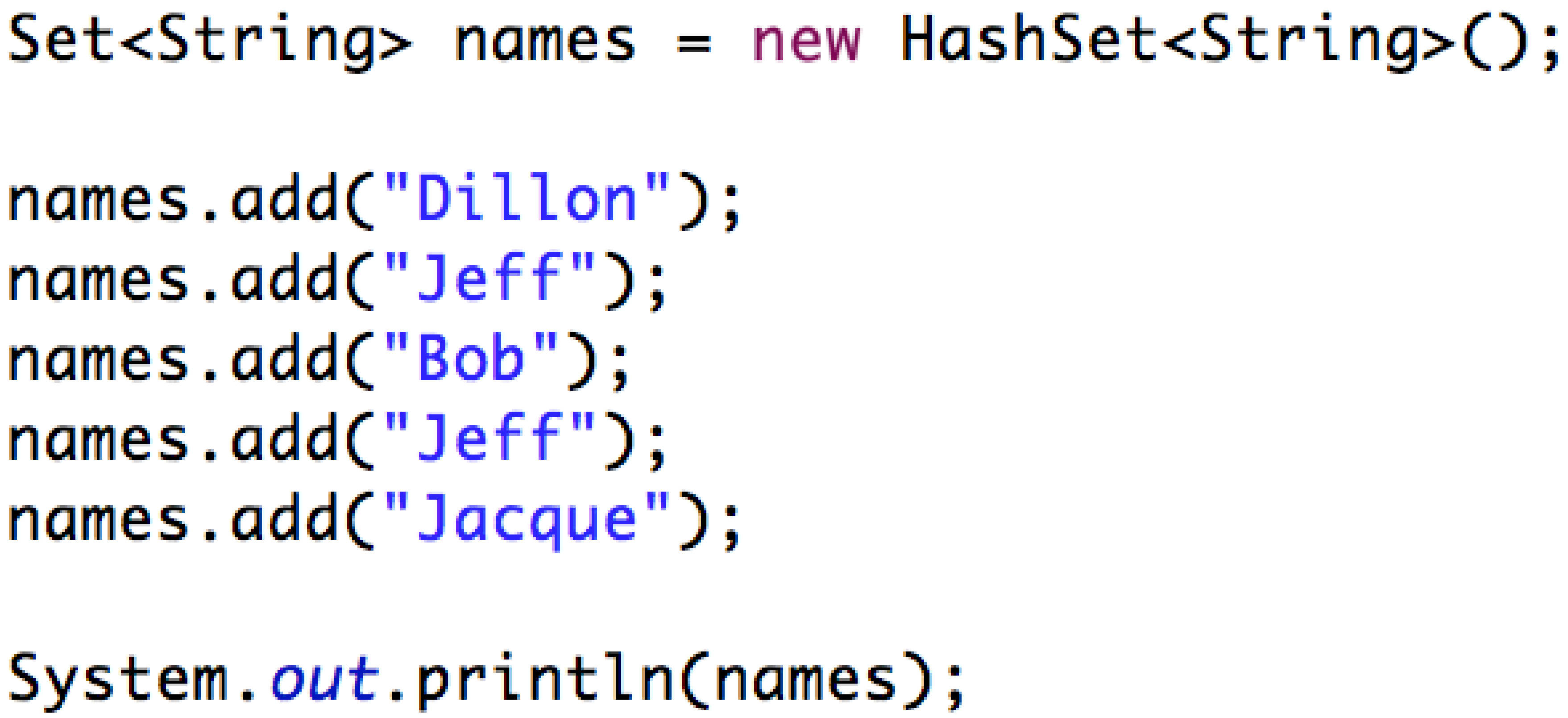
Methods

Modifier and Type	Method and Description
boolean	add(E e) Adds the specified element to this set if it is not already present (optional operation).
boolean	addAll(Collection<? extends E> c) Adds all of the elements in the specified collection to this set if they're not already present (optional operation).
void	clear() Removes all of the elements from this set (optional operation).
boolean	contains(Object o) Returns true if this set contains the specified element.
boolean	containsAll(Collection<?> c) Returns true if this set contains all of the elements of the specified collection.
boolean	equals(Object o) Compares the specified object with this set for equality.
int	hashCode() Returns the hash code value for this set.
boolean	isEmpty() Returns true if this set contains no elements.
Iterator<E>	iterator() Returns an iterator over the elements in this set.
boolean	remove(Object o) Removes the specified element from this set if it is present (optional operation).
boolean	removeAll(Collection<?> c) Removes from this set all of its elements that are contained in the specified collection (optional operation).
boolean	retainAll(Collection<?> c) Retains only the elements in this set that are contained in the specified collection (optional operation).
int	size() Returns the number of elements in this set (its cardinality).
Object[]	toArray() Returns an array containing all of the elements in this set.
<T> T[]	toArray(T[] a) Returns an array containing all of the elements in this set; the runtime type of the returned array is that of the specified array.

Set

- *Set* has many implementations, but *HashSet* is most popular.
- *HashSet* is backed by a hashing table – *we* will look at that later!

Set - Usage



```
Set<String> names = new HashSet<String>();

names.add("Dillon");
names.add("Jeff");
names.add("Bob");
names.add("Jeff");
names.add("Jacque");

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

Prints: “[Jacque, Bob, Dillon, Jeff]”

Output is <u>unordered</u> and “Jeff” only appears once!

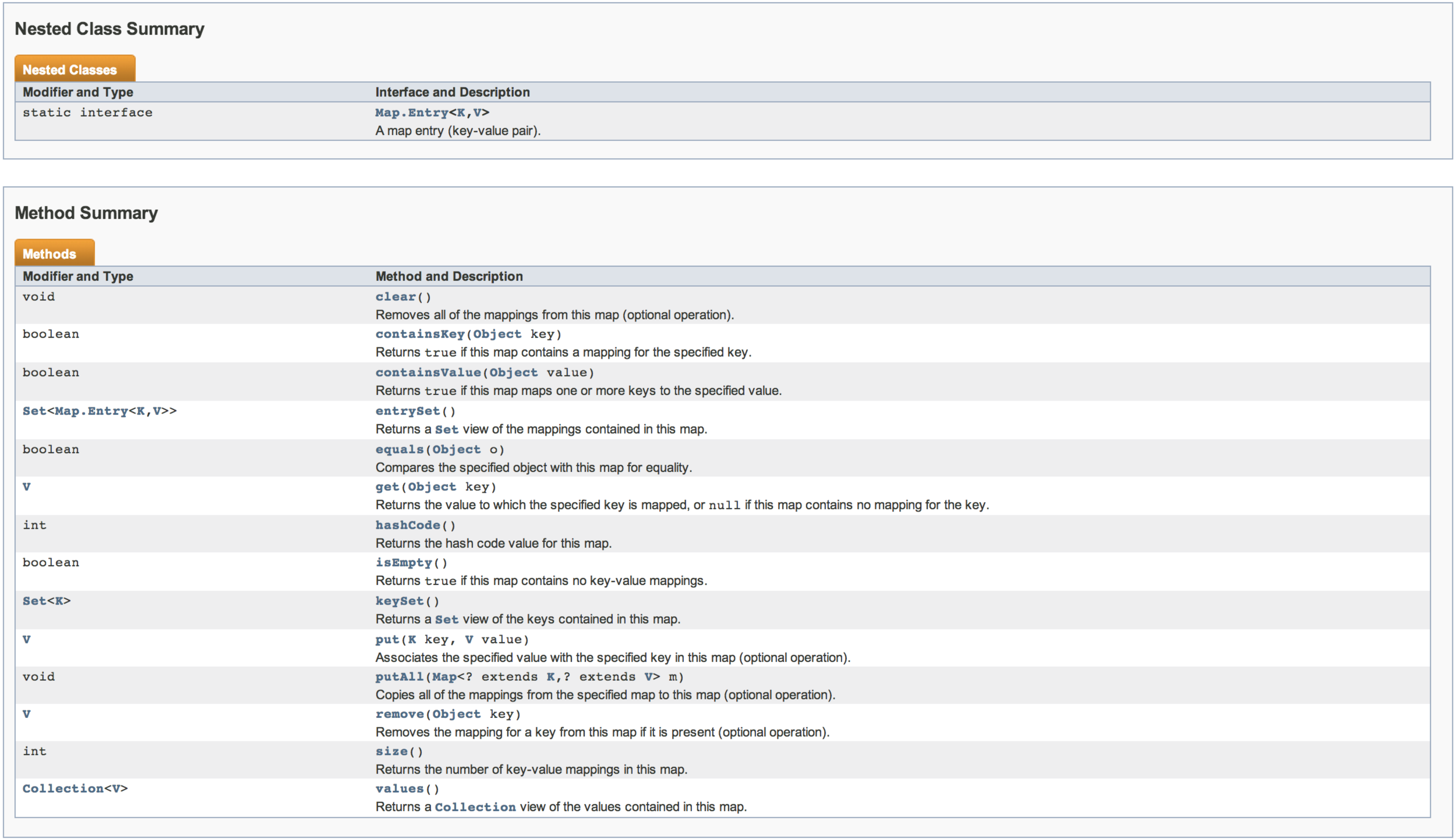
What is set good for?

- For holding a collection of data that is frequently checked for the presence of some element.
- **Example:** A set of student IDs representing students who are registered in a certain class.

Map

- Found under the *java.util.Map* package.
- Represents a *mapping* of one set of data to another.
- Supports adding and removing, among other operations.

The Map Interface



Nested Class Summary

Nested Classes

Modifier and Type	Interface and Description
static interface	Map.Entry<K,V> A map entry (key-value pair).

Method Summary

Methods

Modifier and Type	Method and Description
void	clear() Removes all of the mappings from this map (optional operation).
boolean	containsKey(Object key) Returns true if this map contains a mapping for the specified key.
boolean	containsValue(Object value) Returns true if this map maps one or more keys to the specified value.
Set<Map.Entry<K,V>>	entrySet() Returns a Set view of the mappings contained in this map.
boolean	equals(Object o) Compares the specified object with this map for equality.
V	get(Object key) Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
int	hashCode() Returns the hash code value for this map.
boolean	isEmpty() Returns true if this map contains no key-value mappings.
Set<K>	keySet() Returns a Set view of the keys contained in this map.
V	put(K key, V value) Associates the specified value with the specified key in this map (optional operation).
void	putAll(Map<? extends K,? extends V> m) Copies all of the mappings from the specified map to this map (optional operation).
V	remove(Object key) Removes the mapping for a key from this map if it is present (optional operation).
int	size() Returns the number of key-value mappings in this map.
Collection<V>	values() Returns a Collection view of the values contained in this map.

Map

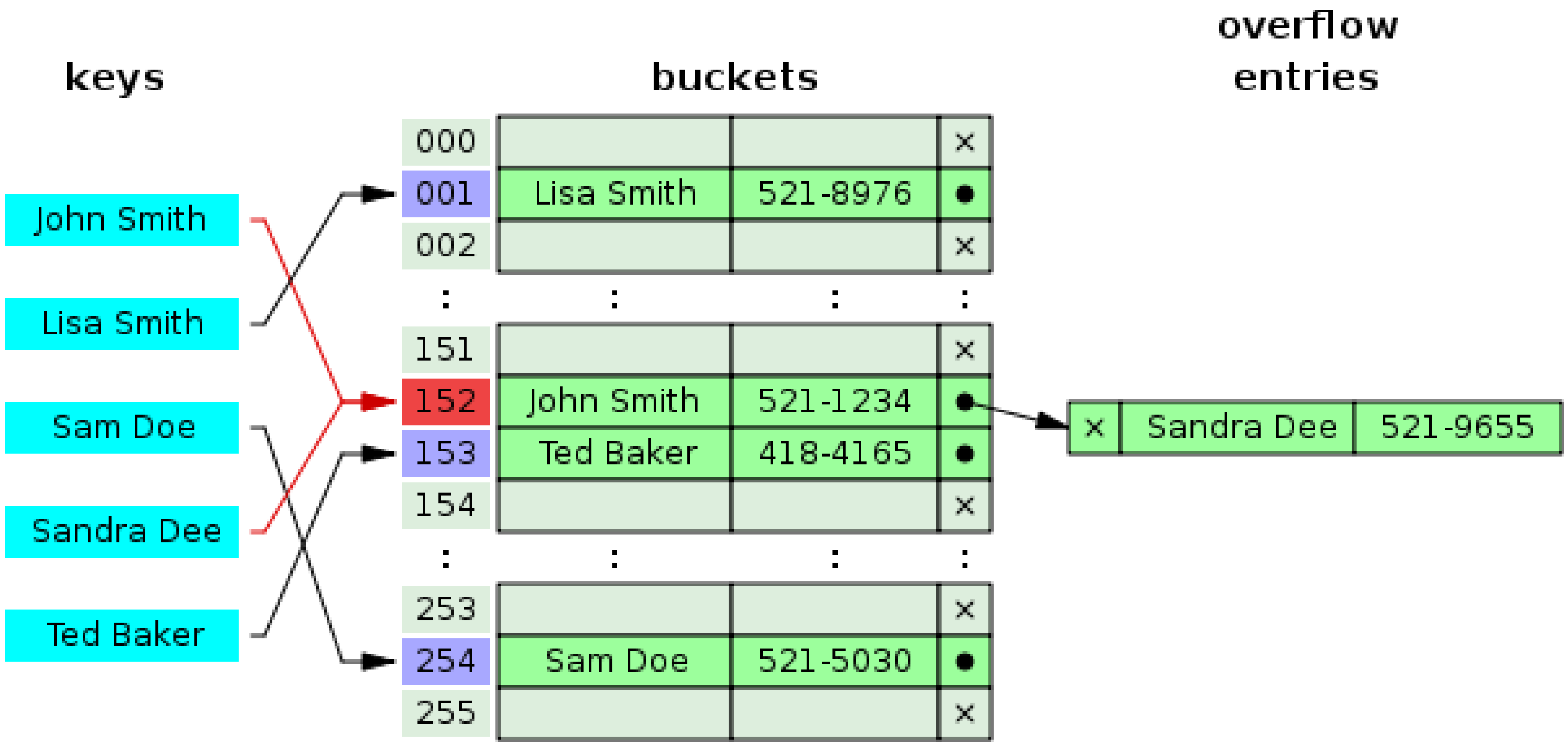
- As we learned in tutorial #6, overriding *hashCode* makes hash-based structures work more efficiently.

- It is not a problem if two different *keys* produce the same hash code, but it is a lot better if they do not!

- Recovering a value using a key has *approximately* constant lookup time, provided we have a good hash function for our keys.

HashMap

- The following maps names to phone numbers.
- Names with the same hash code cause a “collision” and form a chain that must be traversed to find the actual match.



keys

John Smith

Lisa Smith

Sam Doe

Sandra Dee

Ted Baker

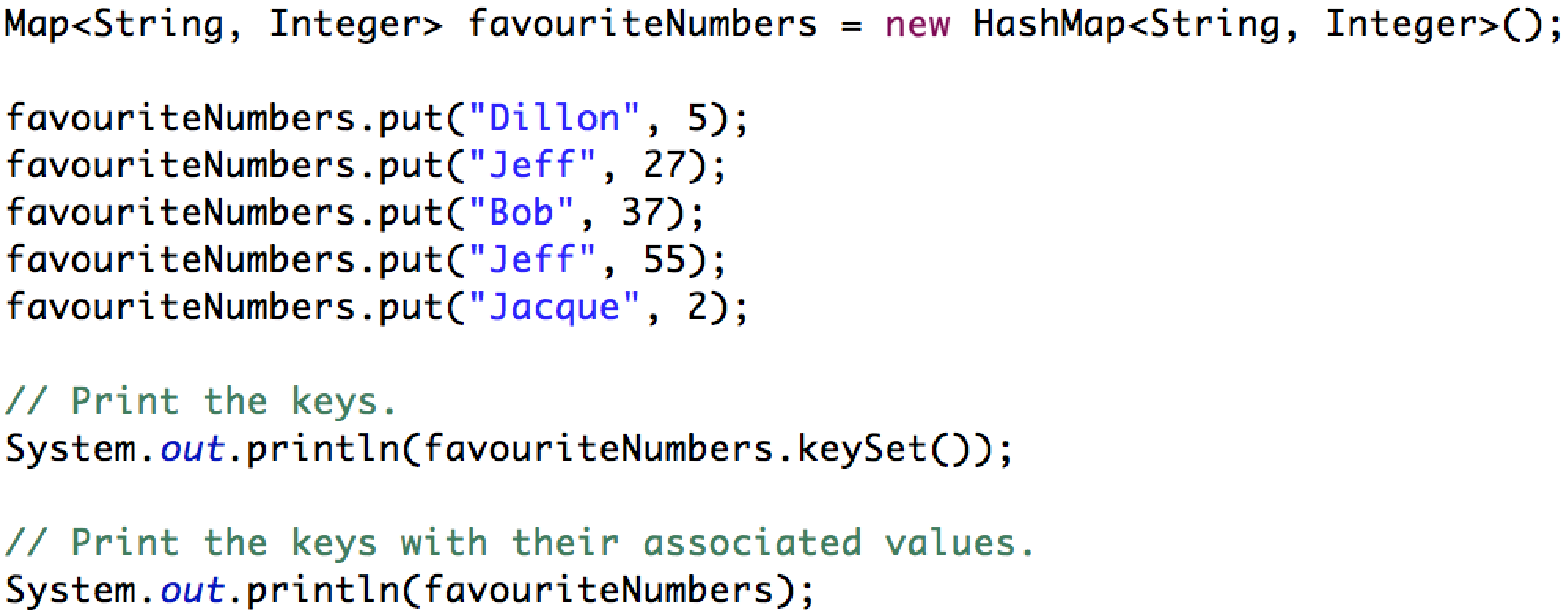
buckets

000			×
001	Lisa Smith	521-8976	●
002			×
:	:	:	:
151			×
152	John Smith	521-1234	●
153	Ted Baker	418-4165	●
154			×
:	:	:	:
253			×
254	Sam Doe	521-5030	●
255			×

overflow entries

×	Sandra Dee	521-9655

Map - Usage



```java
Map<String, Integer> favouriteNumbers = new HashMap<String, Integer>();

favouriteNumbers.put("Dillon", 5);
favouriteNumbers.put("Jeff", 27);
favouriteNumbers.put("Bob", 37);
favouriteNumbers.put("Jeff", 55);
favouriteNumbers.put("Jacque", 2);

// Print the keys.
System.out.println(favouriteNumbers.keySet());

// Print the keys with their associated values.
System.out.println(favouriteNumbers);
```

Prints: “[Jacque, Bob, Dillon, Jeff]”
“{Jacque=2, Bob=37, Dillon=5, Jeff=55}”

When should we use maps?

- Whenever we want to establish a *relationship* between two pieces of data rather than create a collection.
- Because of this <key, value> concept, *Map* is **not** considered part of *Java Collections*!

The *Collections* Family

- *Collections* can be converted between each other.

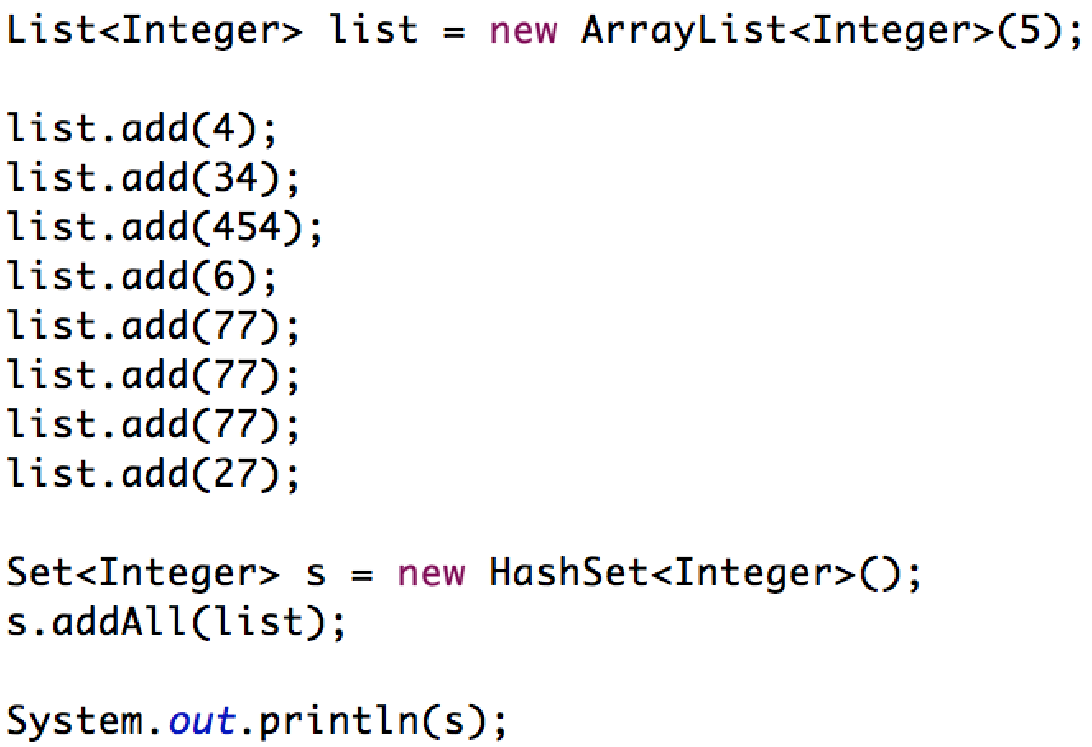
- The ordering that comes about from converting an *unordered* structure to an *ordered* one is essentially *random*.

- The *collections* interface defines the method:

```
addAll(Collection<E> c)
```

The *Collections* Family

- Converting *List* to *Set*

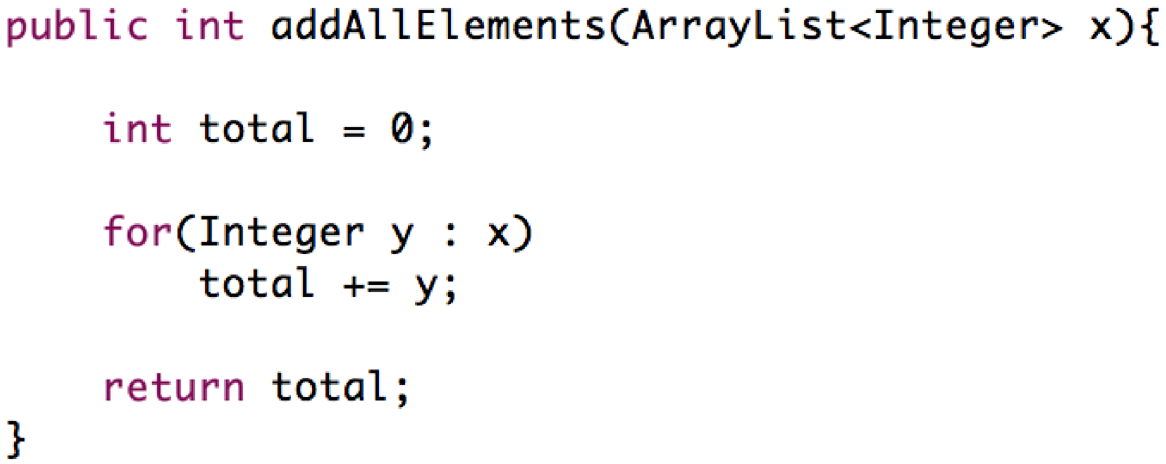


```
List<Integer> list = new ArrayList<Integer>(5);

list.add(4);
list.add(34);
list.add(454);
list.add(6);
list.add(77);
list.add(77);
list.add(77);
list.add(27);

Set<Integer> s = new HashSet<Integer>();
s.addAll(list);

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

Prints: “[34, 4, 6, 77, 454, 27]”

Your interfaces

- The following is a **bad** design decision.

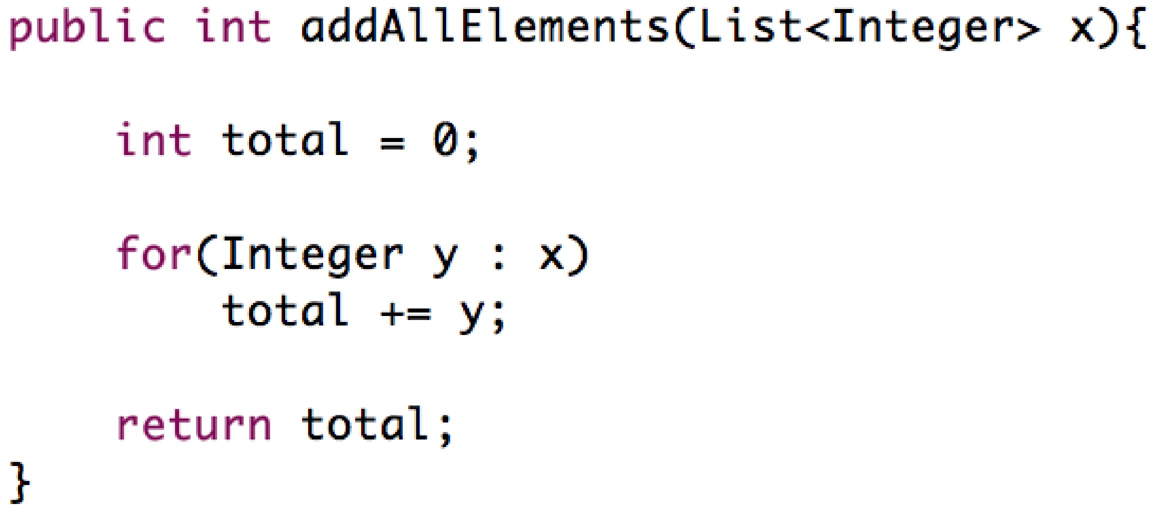


```
public int addAllElements(ArrayList<Integer> x){

    int total = 0;

    for(Integer y : x)
        total += y;

    return total;
}
```

- Why? We are limiting users of our method to *ArrayList* when any implementation of *List* would work as an argument!

Your interfaces

- The following is a **good** design decision.



```
public int addAllElements(List<Integer> x){

    int total = 0;

    for(Integer y : x)
        total += y;

    return total;
}
```

- Users can use this method on any implementation of list!

Your interfaces

- Unless you <u>require</u> a certain implementation of list, you <u>should not</u> require or return anything but the generic interface in your method signature.
- This does not just apply to *List*, but to *every* time you use an implementation of an interface.
- <u>Only</u> expose what <u>needs</u> to be exposed on *your* interface. It will make your code <u>more maintainable</u> in the long run.

Wrapping Up

- What we have seen today is only a subset of all of what is available in *Java Collections*.

- *Collections* has many other useful interfaces and implementations such as *Queue* and *Stack*.

- Make extensive use of *collections* to create powerful and scalable code!

The End

The End :)