Java Collections Framework

Object-Oriented Programming

https://softeng.polito.it/courses/09CBI

SoftEng
http://softeng.polito.it

Version 4.1.0 - April 2020

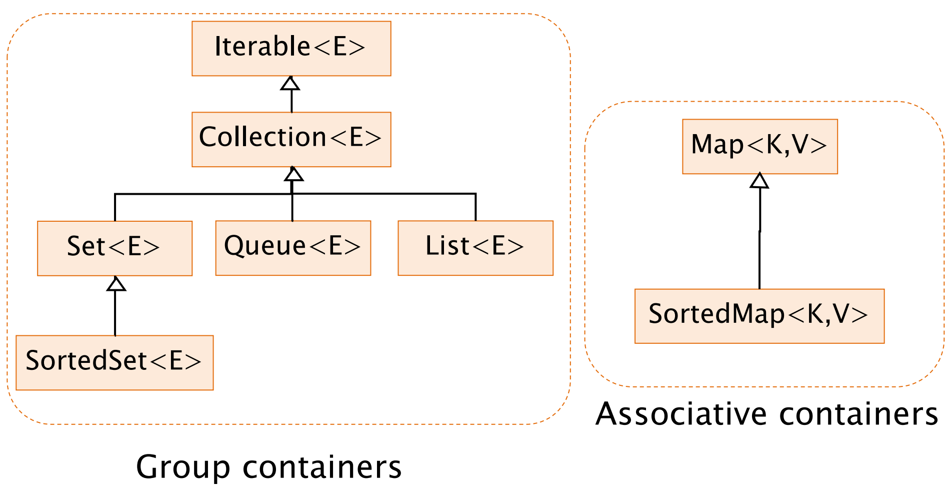
Licensing Note

Collections Framework

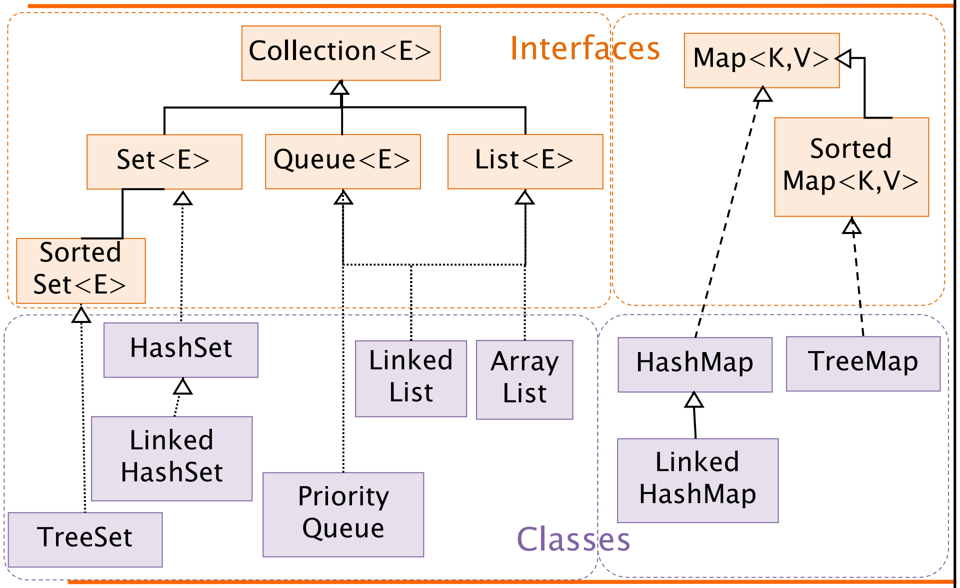
- Interfaces (ADT, Abstract Data Types)
- Implementations (of ADT)
- Algorithms (sort)
- Contained in the package **`java.util`**

- Originally using **`Object`**, since Java 5 redefined as generic

Interfaces



Implementations



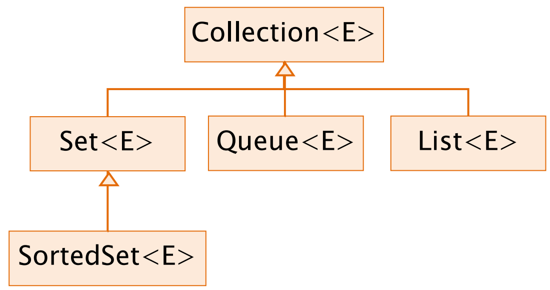
Internals

interface

data structure

	Set	List	Map
Hash table	**HashSet**		**HashMap**
Balanced tree	**TreeSet**		**TreeMap**
Resizable array		**ArrayList**	
Linked list		**LinkedList**	
HT + LL	**LinkedHashSet**		**LinkedHashMap**

classes



GROUP CONTAINERS (COLLECTIONS)

`Collection`

- Group of elements (references to objects)
- It is not specified whether they are
  - Ordered / not ordered
  - Duplicated / not duplicated
- Implements **`Iterable`**
- All classes implementing **`Collection`** shall provide two constructors
  - **`C()`**
  - **`C(Collection c)`**

`Collection` interface

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


`Collection` example

```
Collection<Person> persons =
                new LinkedList<Person>();
persons.add( new Person("Alice") );
System.out.println( persons.size() );

Collection<Person> copy =
                new TreeSet<Person>();
copy.addAll(persons);//new TreeSet(persons)

Person[] array = copy.toArray();
System.out.println( array[0] );
```

`List`

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

`List` interface

```
E get(int index)
E set(int index, E element)
void add(int index, E element)
E remove(int index)

boolean addAll(int index,Collection<E> c)
int indexOf(E o)
int lastIndexOf(E o)
List<E> subList(int from, int to)
```

`List` implementations

- **`ArrayList<E>`**
  - `ArrayList()`
  - `ArrayList(int initialCapacity)`
  - `ArrayList(Collection<E> c)`
  - `void ensureCapacity(int minCapacity)`
- **`LinkedList<E>`**
  - `void addFirst(E o)`
  - `void addLast(E o)`
  - `E getFirst()`
  - `E getLast()`
  - `E removeFirst()`
  - `E removeLast()`

Example

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

l.add(42);          // 42 in position 0
l.add(0, 13);       // 42 moved to position 1
l.set(0, 20);       // 13 replaced by 20
int a = l.get(1);  // returns 42
l.add(9, 30);       // NO: out of bounds
```

IndexOutOfBoundsException

Example II

```
Car[] garage = new Car[20];

garage[0] = new Car();
garage[1] = new ElectricCar();
garage[2] =
garage[3] =

for(int i=0;
    garage[i]
}
```

```
List<Car> garage = new ArrayList<Car>(20);

garage.set( 0, new Car() );
garage.set( 1, new ElectricCar() );
garage.set( 2, new ElectricCar() );
garage.set( 3, new Car());

for(int i; i<garage.size(); i++){
    Car c = garage.get(i);
    c.turnOn();
}
```


Example LinkedList

```
LinkedList<Integer> ll =
            new LinkedList<Integer>();

ll.add(new Integer(10));
ll.add(new Integer(11));

ll.addLast(new Integer(13));
ll.addFirst(new Integer(20));

//20, 10, 11, 13
```

`Queue` interface

- Collection whose elements are inserted using an
  - Insertion order (FIFO)
  - Element order (Priority queue)
- Defines a head position where is the first element that can be accessed
  - **`peek()`**
  - **`poll()`**

`Queue` implementations

- **`LinkedList`**
  - head is the first element of the list
  - FIFO: Fist-In-First-Out
- **`PriorityQueue`**
  - head is the smallest element

Queue example

```
Queue<Integer> fifo =
          new LinkedList<Integer>();
Queue<Integer> pq =
          new PriorityQueue<Integer>();
fifo.add(3); pq.add(3);
fifo.add(1); pq.add(1);
fifo.add(2); pq.add(2);
System.out.println(fifo.peek()); // 3
System.out.println(pq.peek());   // 1
```


Set interface

- Contains no methods
  - Only those inherited from **Collection**
- **add()** has the restriction that no duplicate elements are allowed
  - **e1.equals(e2) == false** $\forall$ e1,e2 $\in \Sigma$
- Iterator
  - The elements are traversed in no particular order

`SortedSet` interface

- No duplicate elements
- Iterator
  - The elements are traversed according to the natural ordering (ascending)
- Augments Set interface
  - `E first()`
  - `E last()`
  - `SortedSet<E> headSet(E toElement)`
  - `SortedSet<E> tailSet(E fromElement)`
  - `SortedSet<E> subSet(E from, E to)`

`Set` implementations

- **`HashSet`** implements **`Set`**
  - Hash tables as internal data structure (faster)
- **`LinkedHashSet`** extends **`HashSet`**
  - Elements are traversed by iterator according to the insertion order
- **`TreeSet`** implements **`SortedSet`**
  - R-B trees as internal data structure (computationally expensive)

Note on sorted collections

- Depending on the constructor used they require different implementation of the custom ordering
- **`TreeSet()`**
  - Natural ordering (elements must be implementations of Comparable)
- **`TreeSet(Comparator c)`**
  - Ordering is according to the comparator rules, instead of natural ordering

Generic collections

- Since Java 5, all collection interfaces and classes have been redefined as Generics
- Use of generics leads to code that is
  - safer
  - more compact
  - easier to understand
  - equally performing

Object list – excerpt

```
public interface List{
  void add(Object x);
  Object get(int i);
  Iterator<E> iterator();
}
public interface Iterator{
  Object next();
  boolean hasNext();
}
```


Example

- Using a list of Integers
  - Without generics (`ArrayList list`)

```
list.add(0, new Integer(42));
int n= ((Integer)(list.get(0))).intValue();
```

  - With generics (`ArrayList<Integer> list`)

```
list.add(0, new Integer(42));
int n= ((Integer)(list.get(0))).intValue();
```

  - + autoboxing (`ArrayList<Integer> list`)

```
list.add(0,new Integer(42));
int n = ((Integer)(list.get(0))).intValue();
```

ITERATORS

`Iterable` interface

- Container of elements that can be iterated upon
- Provides a single instance method:

  ```
  Iterator<E> iterator()
  ```

  - It returns the iterator on the elements of the collection
- Collection extends Iterable

Iterators and iteration

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

`Iterator`

- Allows the iteration on the elements of a collection
- Two main methods:
  - `boolean hasNext()`
    - Checks if there is a next element to iterate on
  - `E next()`
    - Returns the next element and advances by one position
  - `void remove()`
    - Optional method, removes the current element

`Iterator` examples

Print all objects in a list

```
Iterable<Person> persons =
                  new LinkedList<Person>();
…
for(Iterator<Person> i = persons.iterator();
                            i.hasNext(); ) {
   Person p = i.next();
   …
  System.out.println(p);
}
```

31

`Iterator` examples

The for-each syntax avoids
using iterator directly

```
Iterable<Person> persons =
                  new LinkedList<Person>();
…
for(Person p: persons) {
   …
  System.out.println(p);
}
```

32

`Iterator` examples (until Java 1.4)

Print all objects in a list

```
Collection persons = new LinkedList();
…
for(Iterator i= persons.iterator(); i.hasNext(); ) {
    Person p = (Person)i.next();
    …
}
```


Iterable `forEach`

- Iterable defines the default method

  `forEach(Consumer<? super T> action)`
- Can be used to perform operations of elements with a functional interface

```
Iterable<Person> persons;
…
persons.forEach( p -> {
   System.out.println(p);
});
```

Note well

- It is unsafe to iterate over a collection you are modifying (add/remove) at the same time

- Unless you are using the iterator's own methods
  - **`Iterator.remove()`**
  - **`ListIterator.add()`**

35

35

Delete

```
List<Integer> lst=new LinkedList<Integer>();
lst.add(new Integer(10));
lst.add(new Integer(11));
lst.add(new Integer(13));
lst.add(new Integer(20));

int count = 0;
for (Iterator<?> itr = lst.iterator();
                       itr.hasNext(); ) {
   itr.next();
   if (count==1)
      lst.remove(count); // wrong
   count++;
}
```

ConcurrentModificationException

36

36

Delete (cont' d)

```
List<Integer> lst=new LinkedList<Integer>();
lst.add(new Integer(10));
lst.add(new Integer(11));
lst.add(new Integer(13));
lst.add(new Integer(20));

int count = 0;
for (Iterator<?> itr = lst.iterator();
                       itr.hasNext(); ) {
    itr.next();
    if (count==1)
        itr.remove(); // ok
    count++;
}
```

Correct

Add

```
List lst = new LinkedList();
lst.add(new Integer(10));
lst.add(new Integer(11));
lst.add(new Integer(13));
lst.add(new Integer(20));

int count = 0;
for (Iterator itr = lst.iterator();
                    itr.hasNext(); ) {
    itr.next();
    if (count==2)
      lst.add(count, new Integer(22));//wrong
    count++;
}
```

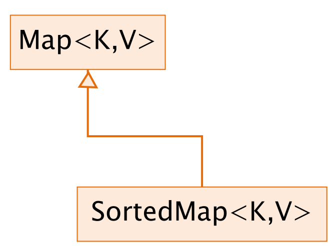
ConcurrentModificationException

Add (cont’d)

```
List<Integer> lst=new LinkedList<Integer>();
lst.add(new Integer(10));
lst.add(new Integer(11));
lst.add(new Integer(13));
lst.add(new Integer(20));

int count = 0;
for (ListIterator<Integer> itr =
     lst.listIterator(); itr.hasNext();){
   itr.next();
   if (count==2)
      itr.add(new Integer(22)); // ok
   count++;
}
```

Correct



ASSOCIATIVE CONTAINERS (MAPS)

`Map`

- A container that associates keys to values (e.g., SSN ⇒ Person)
- Keys and values must be objects
- Keys must be unique
  - Only one value per key

- Following constructors are common to all collection implementers
  - `M()`
  - `M(Map m)`

`Map` interface

- `V put(K key, V value)`
- `V get(K key)`
- `Object remove(K key)`
- `boolean containsKey(K key)`
- `boolean containsValue(V value)`
- `public Set<K> keySet()`
- `public Collection<V> values()`
- `int size()`
- `boolean isEmpty()`
- `void clear()`

Map example: put and get

```
Map<String,Person> people =new HashMap<>();
people.put( "ALCSMT", //ssn
   new Person("Alice", "Smith") );
people.put( "RBTGRN", //ssn
   new Person("Robert", "Green") );

if( ! people.containsKey("RBTGRN"))
  System.out.println( "Not found" );

Person bob = people.get("RBTGRN");

int populationSize = people.size();
```


43

Map ex.: values and keySet

```
Map<String,Person> people =new HashMap<>();
people.put( "ALCSMT", //ssn
  new Person("Alice", "Smith") );
people.put("RBTGRN", //ssn
  new Person("Robert", "Green") );
// Print all people
for(Person p : people.values()){
  System.out.println(p);
}
// Print all ssn
for(String ssn : people.keySet()){
  System.out.println(ssn);
}
```


44

`SortedMap` interface

- The elements are traversed according to the keys' natural ordering
  - Or using comparator passed to ctor
- Augments **`Map`** interface
  - **`SortedMap subMap(K fromKey, K toKey)`**
  - **`SortedMap headMap(K toKey)`**
  - **`SortedMap tailMap(K fromKey)`**
  - **`K firstKey()`**
  - **`K lastKey()`**

`Map` implementations

- Analogous to Set
- **`HashMap`** implements **`Map`**
  - No order
- **`LinkedHashMap`** extends **`HashMap`**
  - Insertion order
- **`TreeMap`** implements **`SortedMap`**
  - Ascending key order

OPTIONAL

Nullability problem

- The typical convention in Java APIs is to let a method return a **`null`** reference to represent the absence of a result.
- The caller must check the return value of the method
- When appropriate checks are not applied, may lead to NPEs

Optional

- **Optional** represents a potential value
- Methods returning **Optional<T>** make explicit that return value may be missing
  - Forces the clients to deal with potentially empty optional

Optional<T>

- Access to embedded value through
  - **boolean isPresent()**
    - checks if Optional contains a value
  - **ifPresent(Consumer<T> block)**
    - executes the given block if a value is present.
  - **T get()**
    - returns the value if present; otherwise it throws a **NoSuchElementException**.
  - **T orElse(T default)**
    - returns the value if present; otherwise it returns a **default** value.
  - **T orElse(Supplier<T> s)**
    - when empty return the value supplied value by **s**

`Optional<T>`

- Creation uses static factory methods:
  - **`of(T v)`**:
    - throw exception if **`v`** is **`null`**
  - **`ofNullable(T v)`**:
    - returns an empty Optional when **`v`** is **`null`**
  - **`empty()`**
    - returns an empty Optional
  - Such methods force the programmer to think about what he’s about to return

Using Collections

Use general interfaces

- E.g. **`List<>`** is better than **`LinkedList<>`**

- General interfaces are more flexible for future changes
- Makes you think
  - First about the type of container
  - Then about the implementation

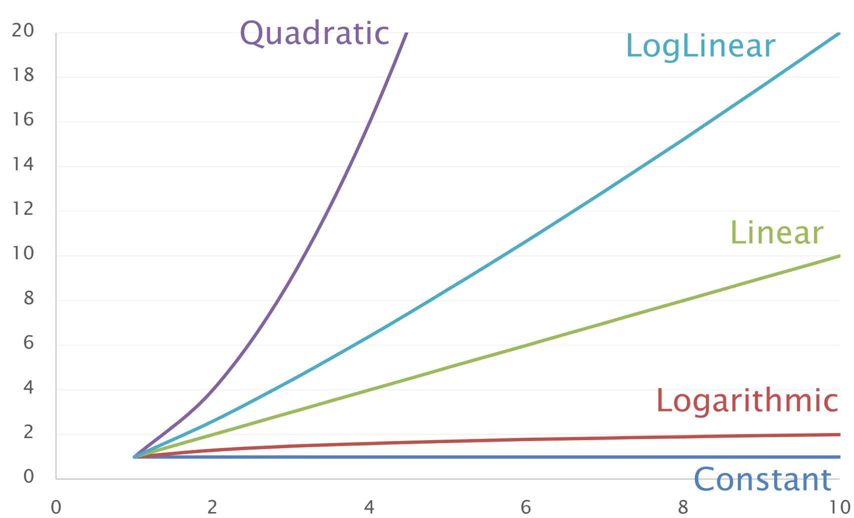
Selecting the container type

- If access by key is needed use a **`Map`**
  - If values sorted by key use a **`SortedMap`**
- Otherwise use a **`Collection`**
  - If indexed access, use a **`List`**
    - Class depends on expected typical operation
  - If access in order, use a **`Queue`**
  - If no duplicates, use a **`Set`**
    - If elements sorted, use a **`SortedSet`**

Efficiency

- Time and Space
- Computed as a function of the number (n) of elements contained
  - Constant: independent of n
  - Logarithmic: grows as $log(n)$
  - Linear: grows proportionally to n
  - Loglinear: grows as $n\ log(n)$
  - Quadratic: grows proportionally to n^2

Efficiency



List implementations

ArrayList

- **get(n)**
  - Constant
- **add(0,…)**
  - Linear
- **add()**
  - Constant

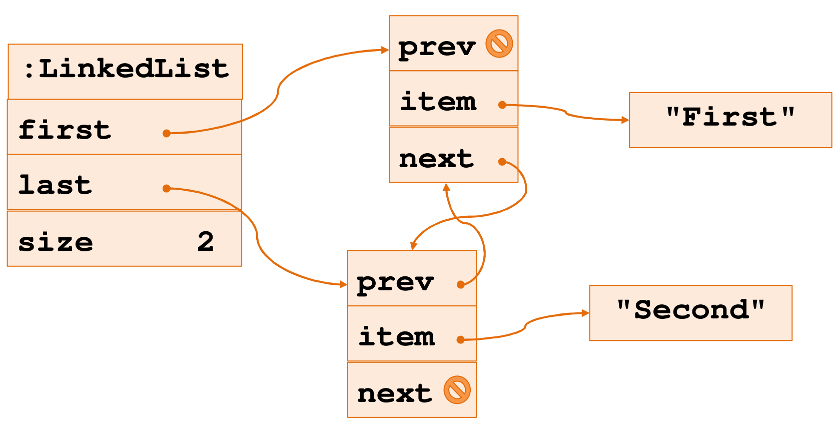
LinkedList

- **get(n)**
  - Linear
- **add(0, …)**
  - Constant
- **add()**
  - Constant

57

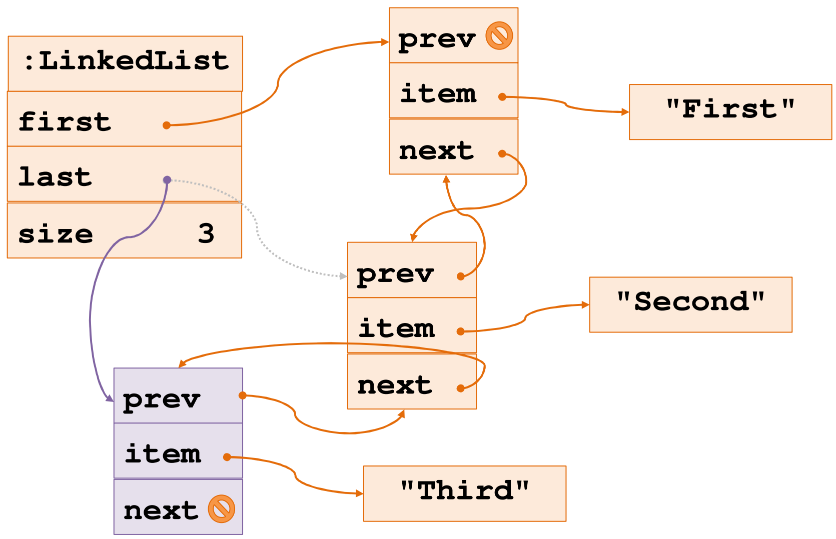
57

Linked list



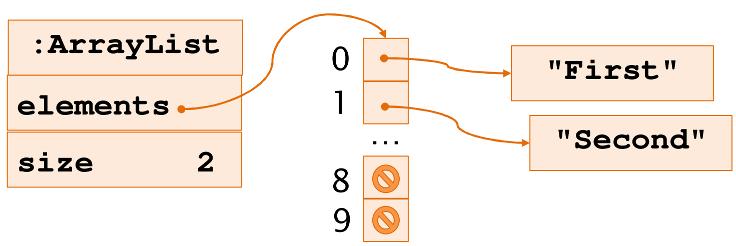
58

Linked list



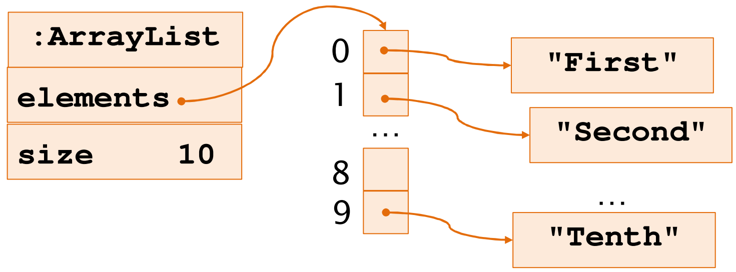
59

Array list

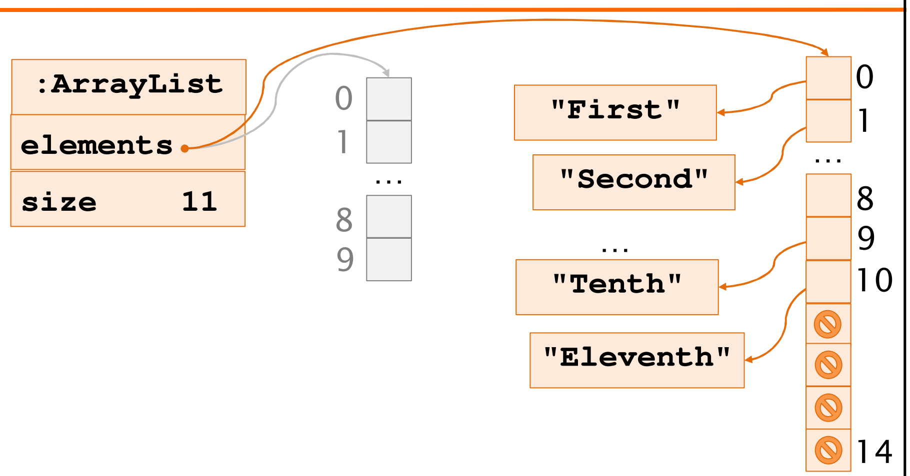


60

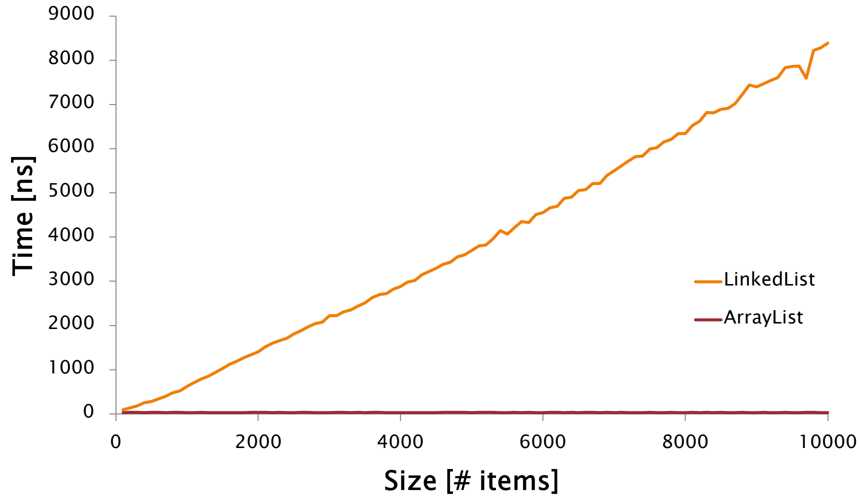
Array list



Array list

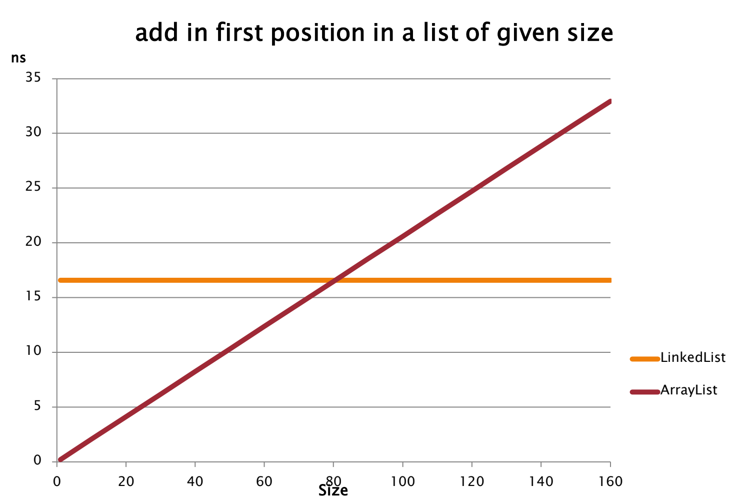


`List` implementations – Get



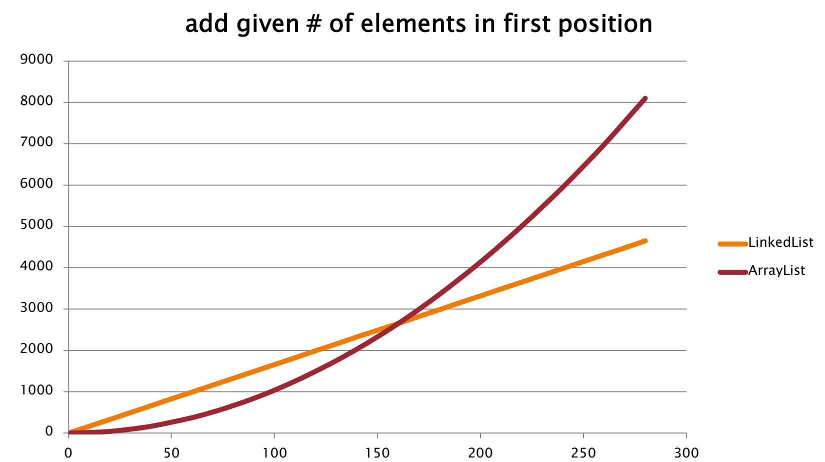
63

`List` Implementations – Add



64

`List` Implementations – Add



`List` implementation – Models

	`LinkedList`	`ArrayList`
Add in first pos. in list of size *n*	$t(n) = C_L$	$t(n) = n \cdot C_A$
Add *n* elements	$t(n) = n \cdot C_L$	$t(n) = \sum_{i=1}^{n} C_A \cdot i = \frac{C_A}{2} n \cdot (n-1)$

$C_L = 16.0$ ns
$C_A = 0.2$ ns

Using maps

- Getting an item

```
String val = map.get(key);
if( val == null ) {
   // not found
}
```

- Or

```
if( ! map.containsKey(key)) {
   // not found
}
String val = map.get(key);
```


Using maps

- Updating entries
  - E.g. counting frequencies

```
Map<String,Integer> wc=new XMap<>();
for(String w : words) {
  Integer i= wc.get(w);
  wc.put(w, i==null?1:i+1);
}
```

Using maps

- Updating entries
  - E.g. counting frequencies

```
Map<String,Integer> wc=new XMap<>();
for(String w : words) {
  wc.compute(w, (k,v)->v==null?1:v+1);
}
```

Autoboxing hides memory fee of 16 bytes per increment due to object creation:
`new Integer(v.intValue()+1)`

Using maps

- Updating entries
  - E.g. counting frequencies

```
class Counter {
  int i=0;
}
```

```
Map<String,Integer> wc=new XMap<>();
for(String w : words) {
  wc.computeIfAbsent(w,
                k->new Counter()).i++;
}
```

~40% faster than with `Integer` – 16 bytes per each increment

Using maps

- Keeping items sorted
  - Using sorted maps

```
SortedMap<…> wc=new TreeMap<>();
```

  - “A”=1, “All”=3, “And"=2, “Barefoot”=1,…

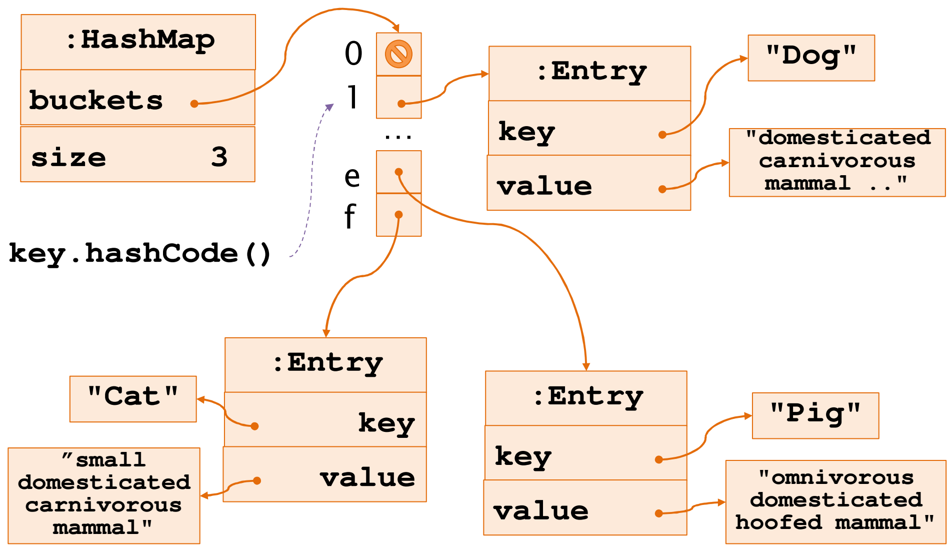
```
Map<…> wc=new HashMap<>();
```

  - “reason”=1, “been”=1, “spoke”=1, “let”=1

HashMap

- Get/put takes constant time (in case of no collisions)
- Automatic re-allocation when load factor reached
- Constructor optional arguments
  - load factor (default = .75)
  - initial capacity (default = 16)

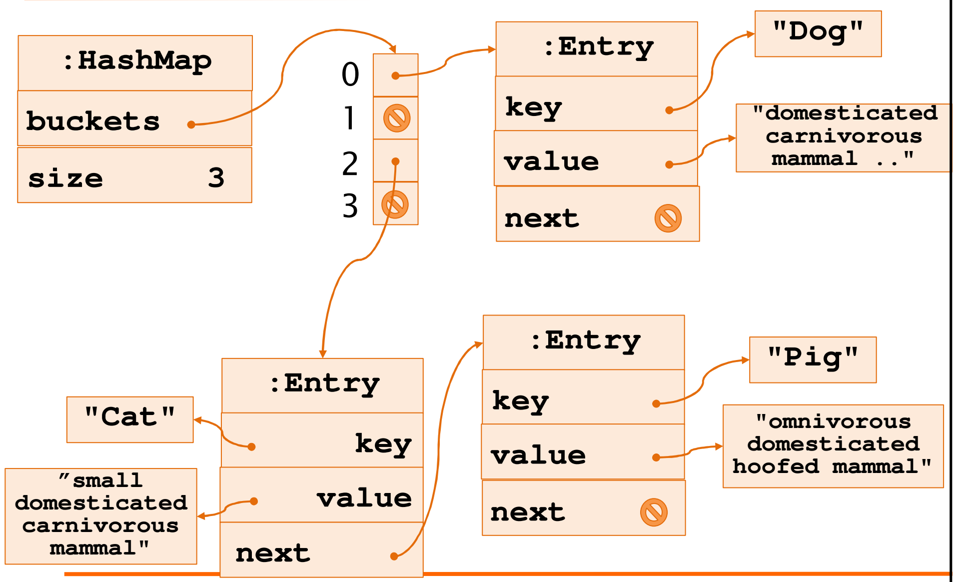
Hashmap



Hash limitations

- Hash based containers **`HashMap`** and **`HashSet`** work better if entries define a suitable **`hashCode()`** method
  - Values must be as spread as possible
  - Otherwise collisions occur
    - When two entries fall in the same bucket
    - In such a case elements are chained in a list
    - Chaining reduces time efficiency

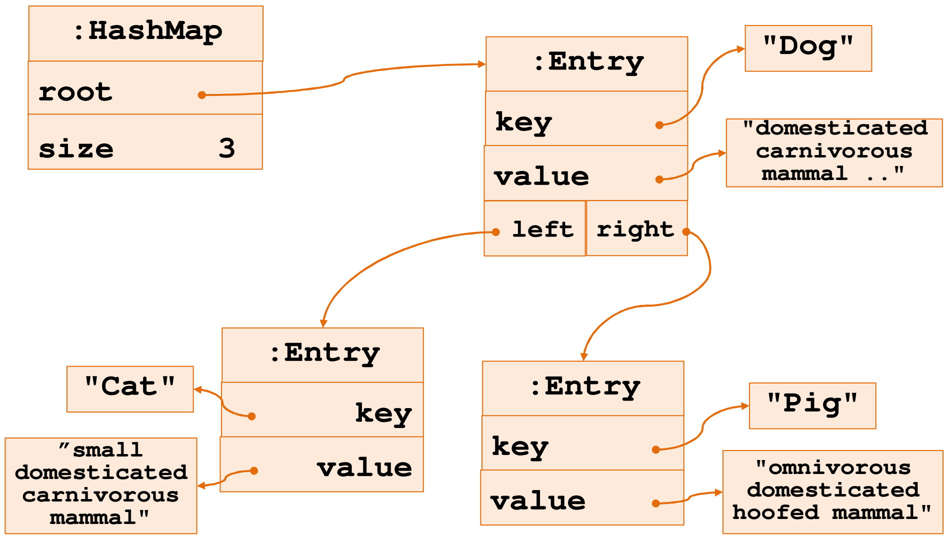
Hashmap (chaining)



TreeMap

- Based on a Red-Black tree
- Get/put takes log time
- Keys are maintained and will be traversed in order
  - Key class must be **`Comparable`**
  - Or a **`Comparator`** must be provided to the constructor

TreeMap



Tree limitations

- Tree based containers (**`TreeMap`** and **`TreeSet`**) require either
  - Entries with a natural order (**`Comparable`**)
  - A **`Comparator`** to sort entries
- **`TreeMap`** keeps keys sorted, and return values sorted by key

Search efficiency

- Example:
  - 100k searches in a container require

size	HashMap	TreeMap	ArrayList	LinkedList
100k	3ms	60ms	40s	>1h
200k	3ms	65ms	110s	

ALGORITHMS

Algorithms

- Static methods of **`java.util.Collections`**
  - Work on List since it has the concept of position
- **`sort`**() – merge sort of List, $n \log(n)$
- **`binarySearch`**() – requires ordered sequence
- **`shuffle`**() – unsort
- **`reverse`**() – requires ordered sequence
- **`rotate`**() – of given a distance
- **`min`**(), **`max`**() – in a Collection

81

`sort()` method

- Operates on **`List<T>`**
  - Require access by index to perform sorting
- Two variants:

```
<T extends Comparable<? super T>>
void sort(List<T> list)
```

```
<T> void sort(List<T> list,
        Comparator<? super T> cmp)
```


82

Sort generic

~~T~~ extends Comparable<~~?~~ super ~~T~~>

MasterStudent **Student** **MasterStudent**

- Why `<? super T>` instead of just `<T>` ?
  - Suppose you define
    - `MasterStudent extends Student { }`
  - Intending to inherit the Student ordering
    - It does not implement `Comparable<MasterStudent>`
    - But `MasterStudent` extends (indirectly) `Comparable<Student>`

Search

- `<T> int binarySearch(List<? extends Comparable<? super T>> l, T key)`
  - Searches the specified object
  - List must be sorted into ascending order according to natural ordering
- `<T> int binarySearch(List<? extends T> l, T key, Comparator<? super T> c)`
  - Searches the specified object
  - List must be sorted into ascending order according to the specified comparator

Wrap-up

- The collections framework includes interfaces and classes for containers
- There are two main families
  - Group containers
  - Associative containers
- All the components of the framework are defined as generic types