Java Collections Framework

http://softeng.polito.it

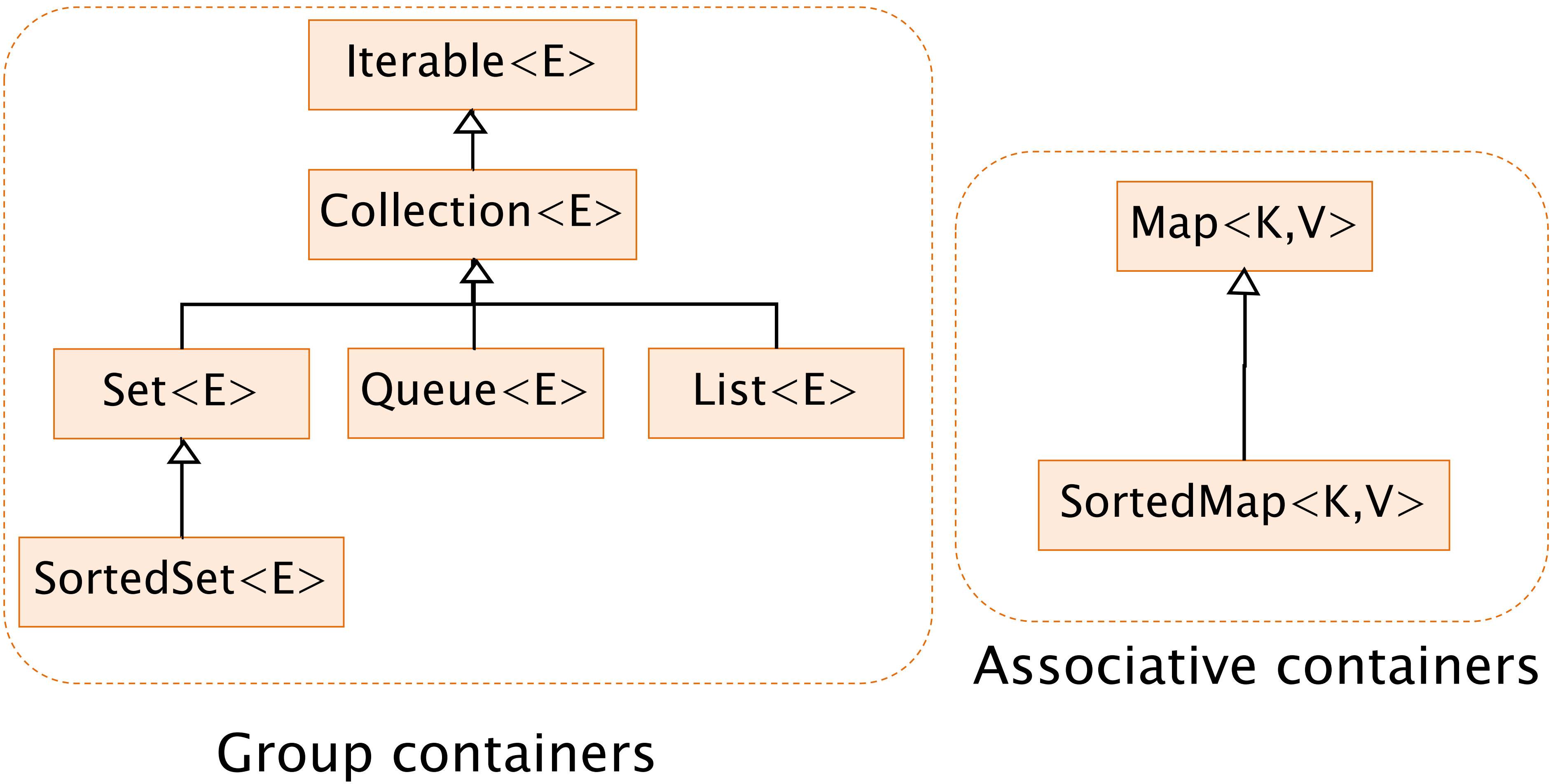
Version 3.2.1 - April 2016

Framework

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

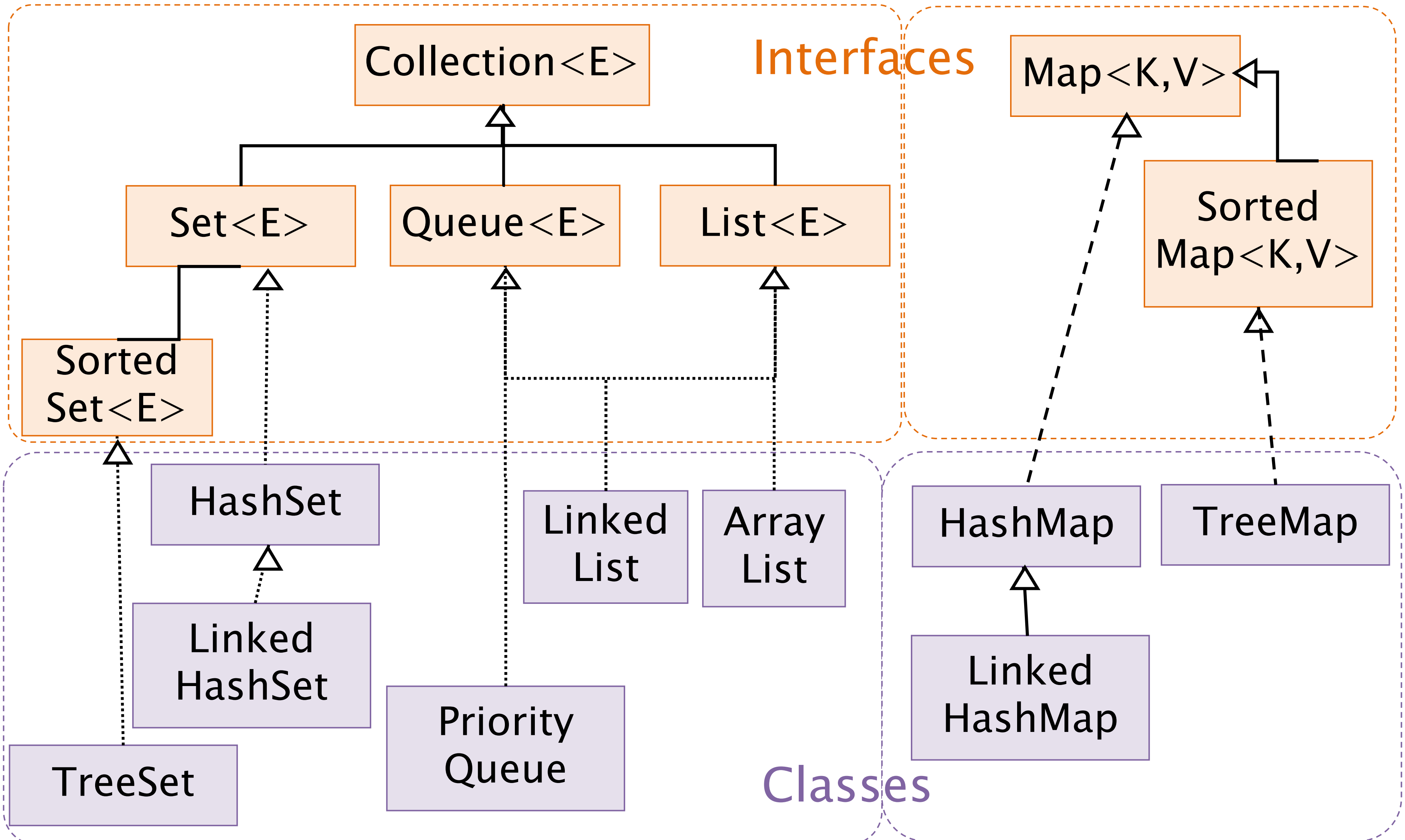
- Originally using Object, since Java 5 redefined as generic

Interfaces



Iterable<E>
Collection<E>
Set<E>
Queue<E>
List<E>
SortedSet<E>
Map<K,V>
SortedMap<K,V>
Associative containers
Group containers

Implementations



Internals

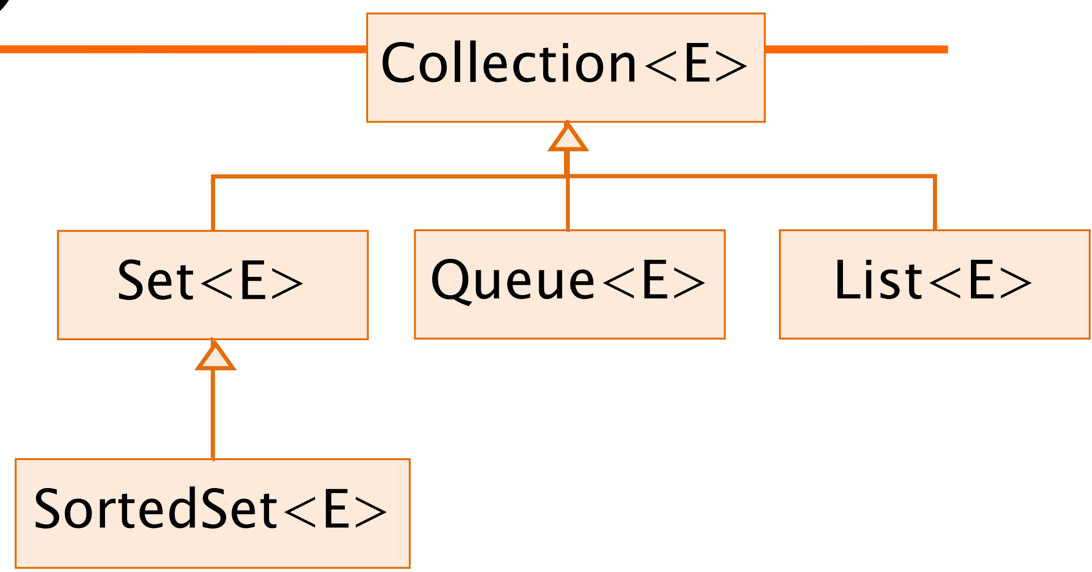
data structure

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

interface

classes

Group containers (Collections)



SoftEng
http://softeng.polito.it

Collection

- Group of elements (references to objects)
- It is not specified whether they are
  - Ordered / not ordered
  - Duplicated / not duplicated
- Implements Iterable
- The following constructors are common to all classes implementing Collection
  - `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] );
```

SoftEng http://softeng.polito.it

List

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

`List` specific methods

```
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

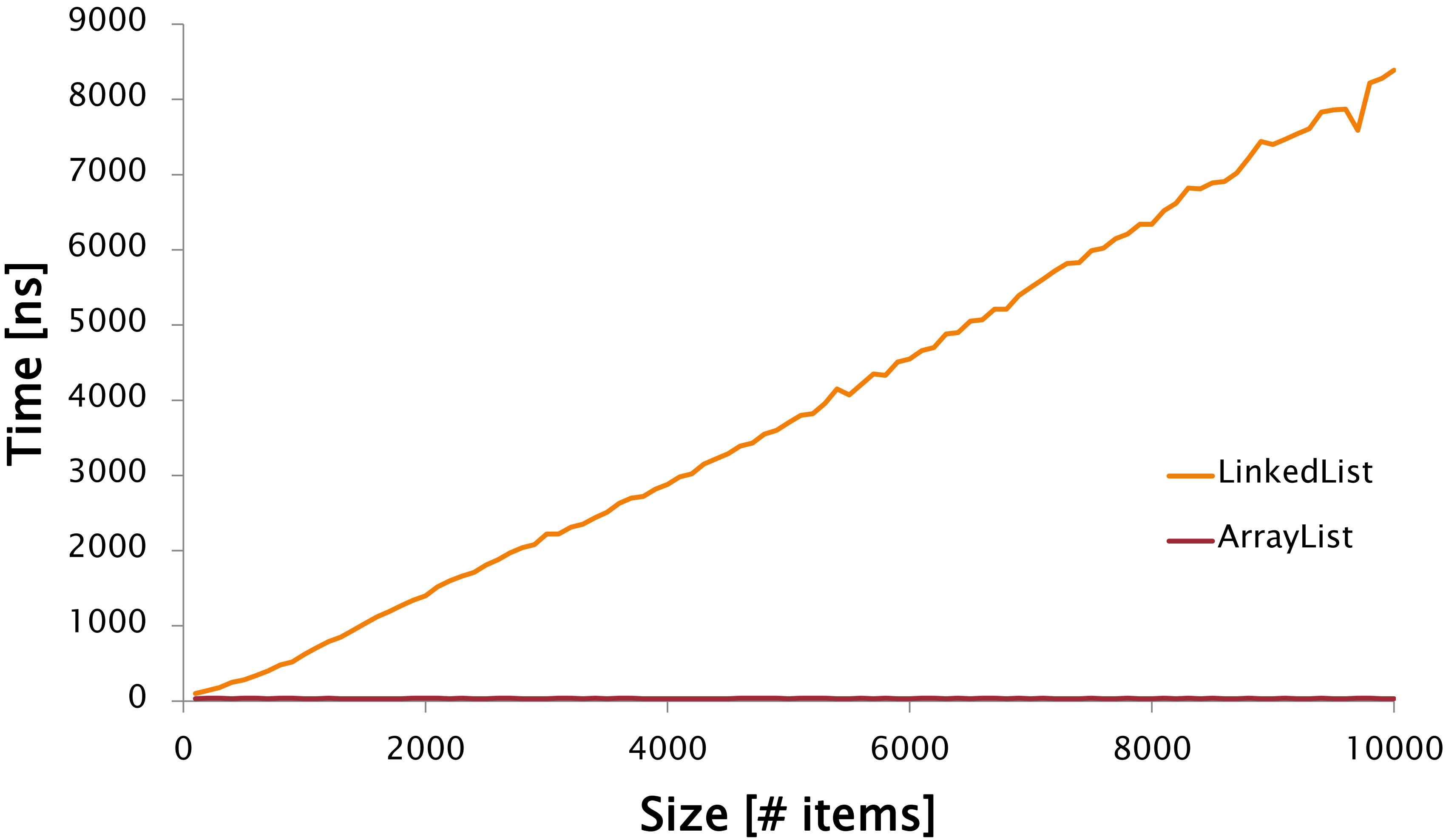
`ArrayList`

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

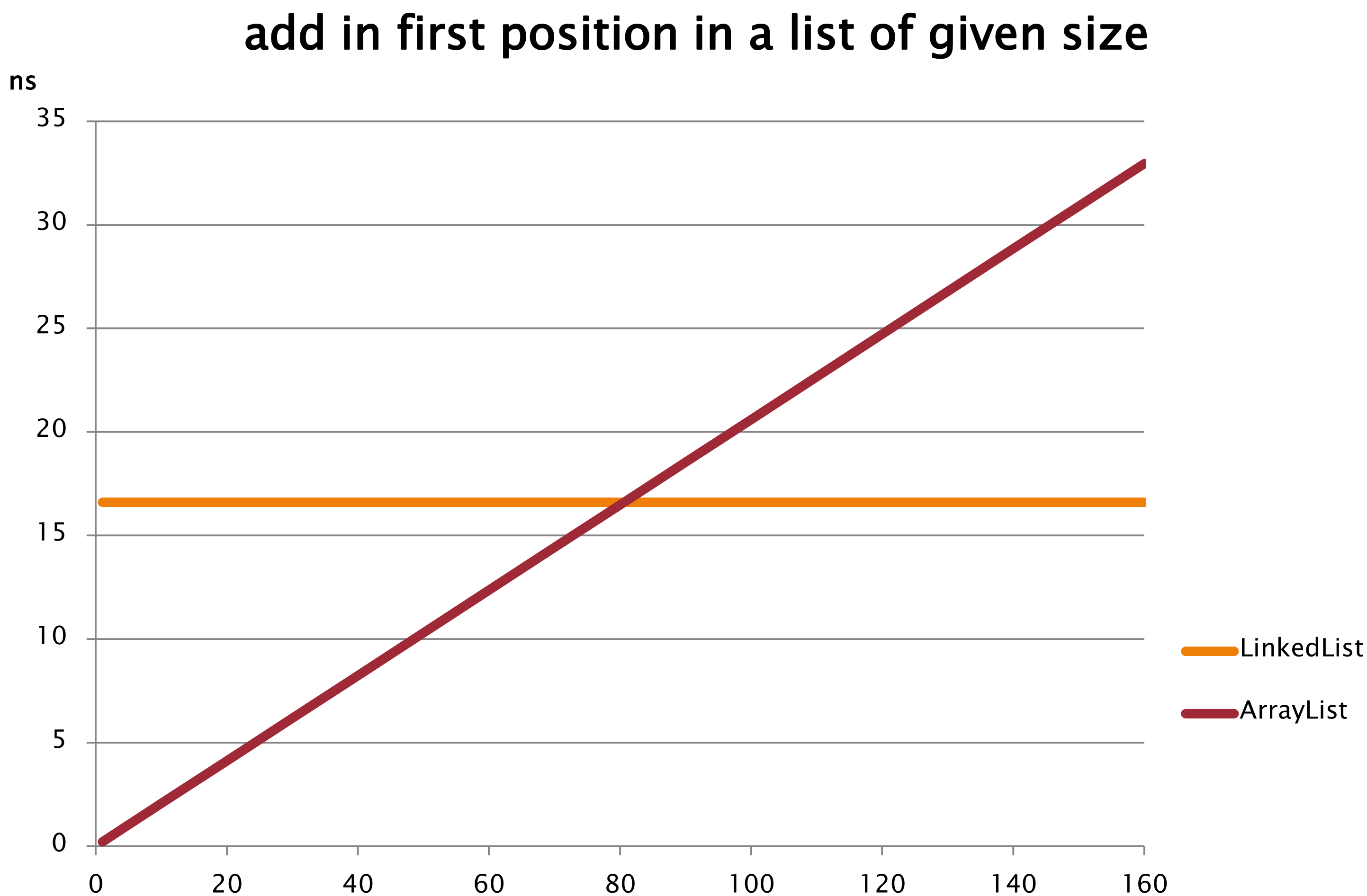
`LinkedList`

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

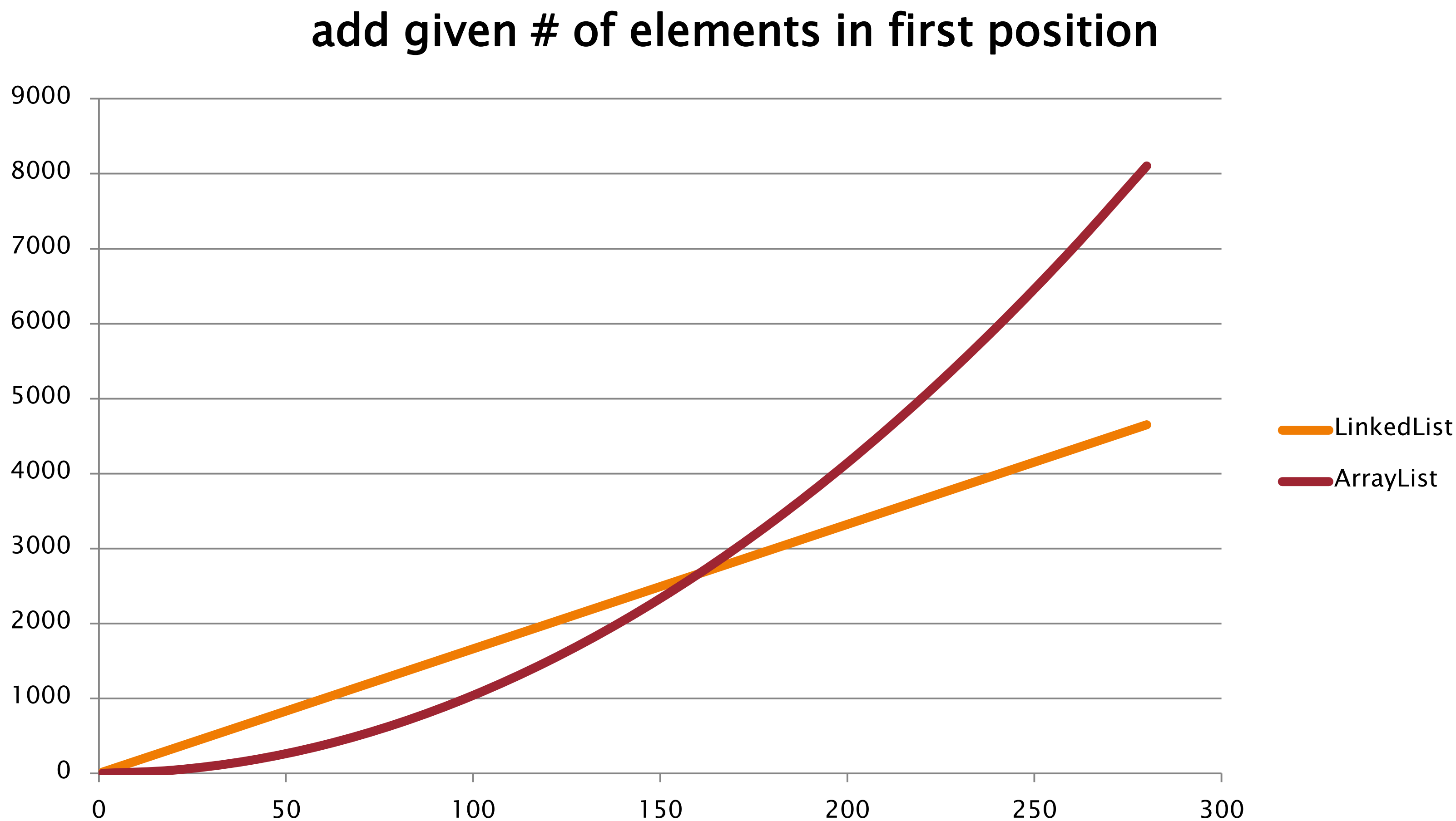
`List` implementations – Get



`List` Implementations – Add



`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

`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 I

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

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

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

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

Example II

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

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

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

```
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 III

```
List l = new ArrayList(2); // 2 refs to null

l.add(new Integer(11));    // 11 in position 0
l.add(0, new Integer(13)); // 11 in position 1
l.set(0, new Integer(20)); // 13 replaced by 20

l.add(9, new Integer(30)); // NO: out of bounds
l.add(new Integer(30));    // OK, size extended
```

`Queue` interface

- Collection whose elements have an order
  - not and ordered collection though
- Defines a head position where is the first element that can be accessed
  - **`peek()`**
    - **`Retrieves, but does not remove, the head of this queue`**
  - **`poll()`**
    - **`Retrieves and removes the head of this queue`**

`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
  - `Object first()`
  - `Object last()`
  - `SortedSet headSet(Object toElement)`
  - `SortedSet tailSet(Object fromElement)`
  - `SortedSet subSet(Object from, Object 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 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);
}
```

`Iterator` examples

The for-each syntax avoids using iterator directly

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

`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();
    …
}
```

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()`

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

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

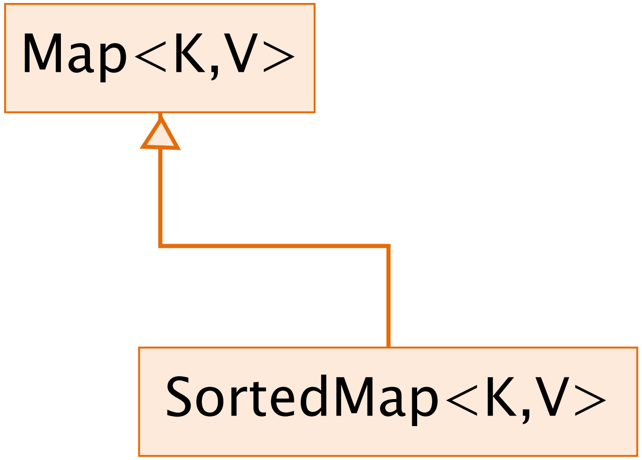
Add (cont'd)

```
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` interface

- 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

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

Person bob = people.get(“RBTGRN”);
if( bob == null )
  System.out.println( “Not found” );

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

`SortedMap` interface

- The elements are traversed according to the keys' natural ordering (ascending)
- 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

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)

Using `HashMap`

```
Map<String,Student> students =
        new HashMap<String,Student>();

students.put("123",
        new Student("123","Joe Smith"));

Student s = students.get("123");

for(Student si: students.values()){
    …
}
```

TreeMap

- Get/put takes log time
- Based on a Red-Black tree
- Keys are maintained and will be traversed in order
- Constructor optional arguments
  - Comparator to replace the natural order of keys

ALGORITHMS

Algorithms

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

`sort()` method

- Operates on `List<T>`
  - Require access by index to perform sorting
- Two generic overloads:
  - on `Comparable` objects:

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

  - using a `Comparator` object:

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

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>`

Custom ordering (alternative)

```
List students = new LinkedList();

students.add(new Student("Mary","Smith",34621));
students.add(new Student("Alice","Knight",13985));
students.add(new Student("Joe","Smith",95635));

Collections.sort(students); // sort by name

Collections.sort(students,
new StudentIDComparator()); // sort by ID
```

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

Algorithms – Arrays

- Static methods of **`java.util.Arrays`** class
  - Work on object arrays

- **`sort`**()
- **`binarySearch`**()

Search – Arrays

- `int binarySearch(Object[] a, Object key)`
  - Searches the specified object
  - Array must be sorted into ascending order according to natural ordering
- `int binarySearch(Object[] a, Object key, Comparator c)`
  - Searches the specified object
  - Array 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