Generics

Generics

- “Generic” is a built-in language feature.
- Enable generic algorithms
- It helps reducing run-time bugs possibly occurred when some classes/interfaces are to be used with any types of objects (the *Object* class).

A Box of “Any object”

Box

Object item;

void putIn(Object)

Object takeOut()

```
public class Box{
    Object item =null;
    public void putIn(Object o){
        item =o;
    }
    public Object takeOut(){
        return item;
    }
}
```

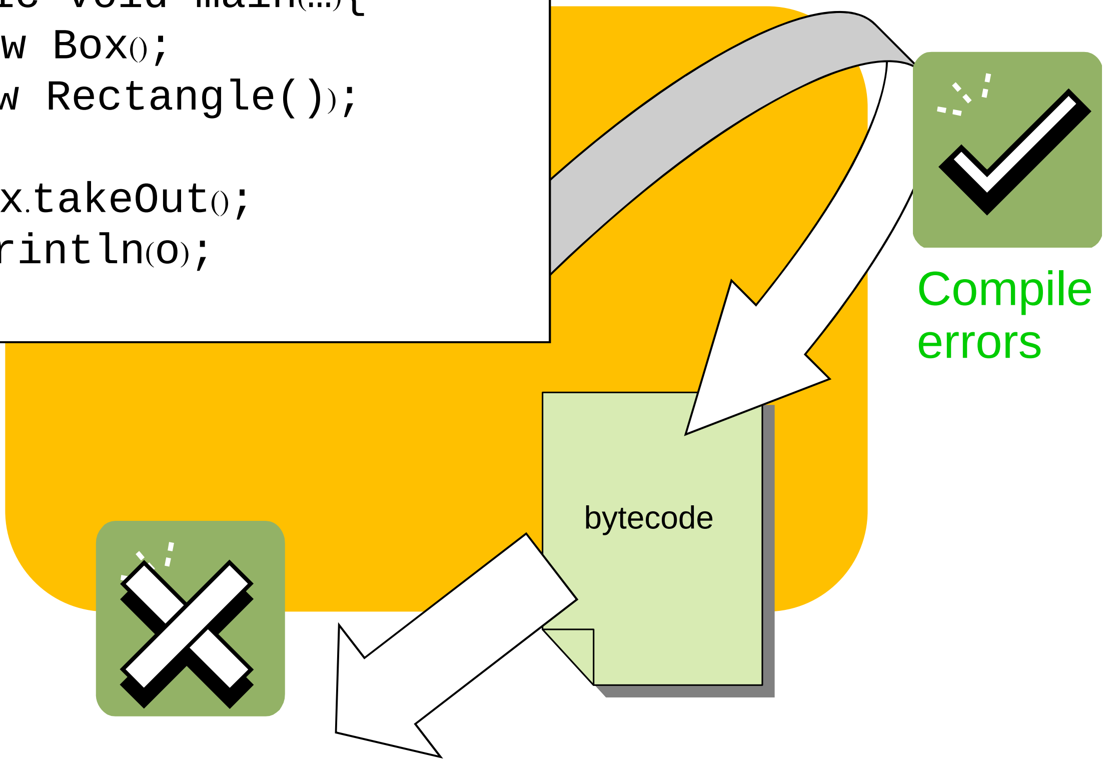
Write a program that puts an object of type Integer into the box and take it out into a variable

```
public static void main(…){
    Box box = new Box();
    box.putIn(new Rectangle());
    Integer o =(Integer) box.takeOut();
    System.out.println(o);
}
```

Compile-time Error
???

Generics (not yet)

```
public static void main(…){
 Box box = new Box();
 box.putIn(new Rectangle());
 Integer o =
  (Integer)box.takeOut();
 System.out.println(o);
}
```



Compile with no errors

```
---------- JVM ----------
Exception in thread "main" java.lang.ClassCastException:
java.awt.Rectangle cannot be cast to java.lang.Integer
              at Box.main(Box.java:13)
```

Run-time Error

Generics

```
public class Box <T>{
  T item = null;
  public void putIn(T o){
       item = o;
  }
  public T takeOut(){
       return item;
  }
}
```

If this is not an Integer object, the compiler will not allow. This way, the programmer knows at compile-time.

```
public static void main(String [] args){
 Box <Integer> box = new <Integer> Box();
 box.putIn(new Integer(88));
 Integer o =
  (Integer)box.takeOut();
 System.out.println(o);
}
```

Naming Convention

- Single, uppercase character
- The most commonly used type parameter names are:
  - ***E*** : Element (used extensively by the Java Collections Framework)
  - ***K*** : Key
  - ***N*** : Number
  - ***T*** : Type
  - ***V*** : Value
- Used throughout the Java SE API

Generic Type Wildcards

`<? extends T>`

An unknown type that is a subclass of *T*, possibly *T* itself

`<? super T>`

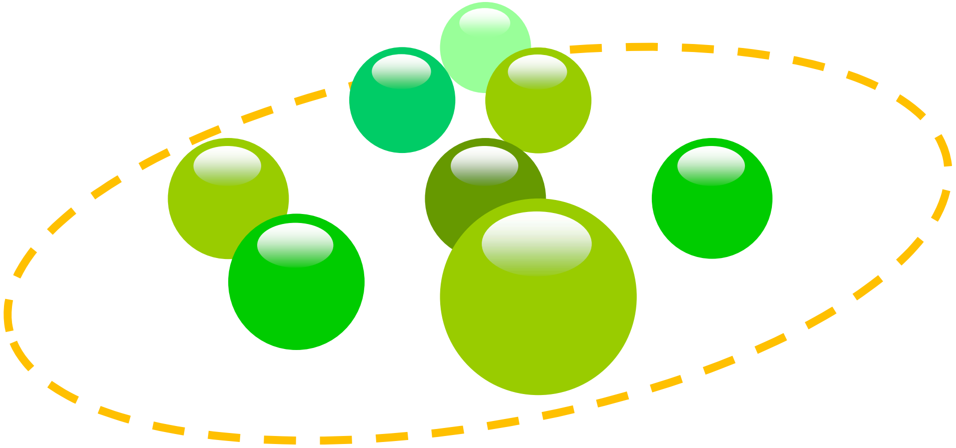
An unknown type that is a superclass of *T*, possibly *T* itself

`<?>`

An unknown type

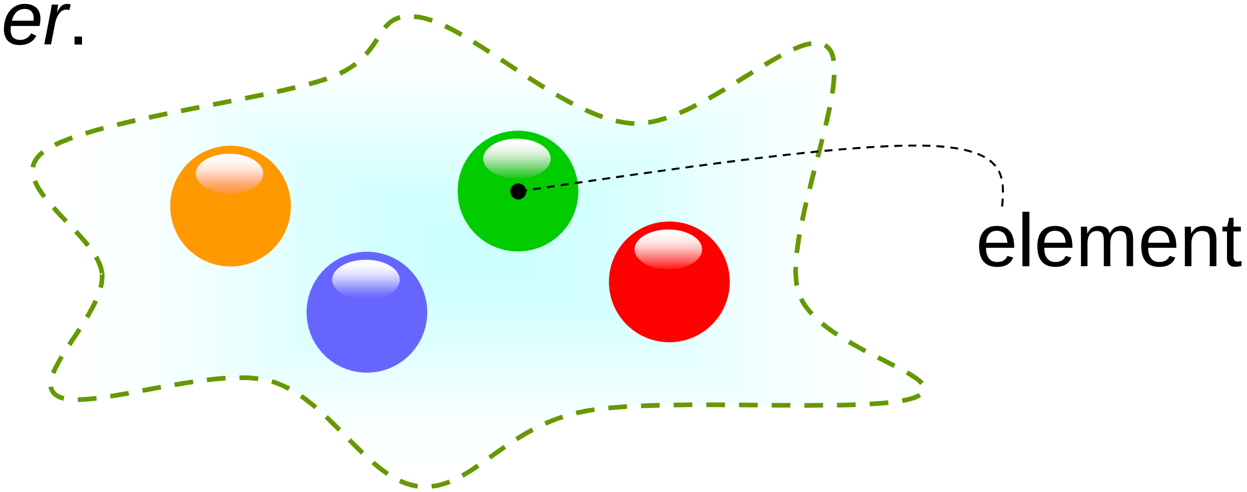
(i.e. `<? extends Object>`)

Collections Framework



Java Collections Framework

- A *collection* is an *object that groups multiple elements* into a single unit, sometime called a *container*.



- The *Collections Framework* provides a well-designed set of interfaces and classes for storing and manipulating groups of data.

Interfaces/Implementations/Algorithms

Interfaces

- Abstract data type
- Allow collections to be manipulated independently of the details of implementation

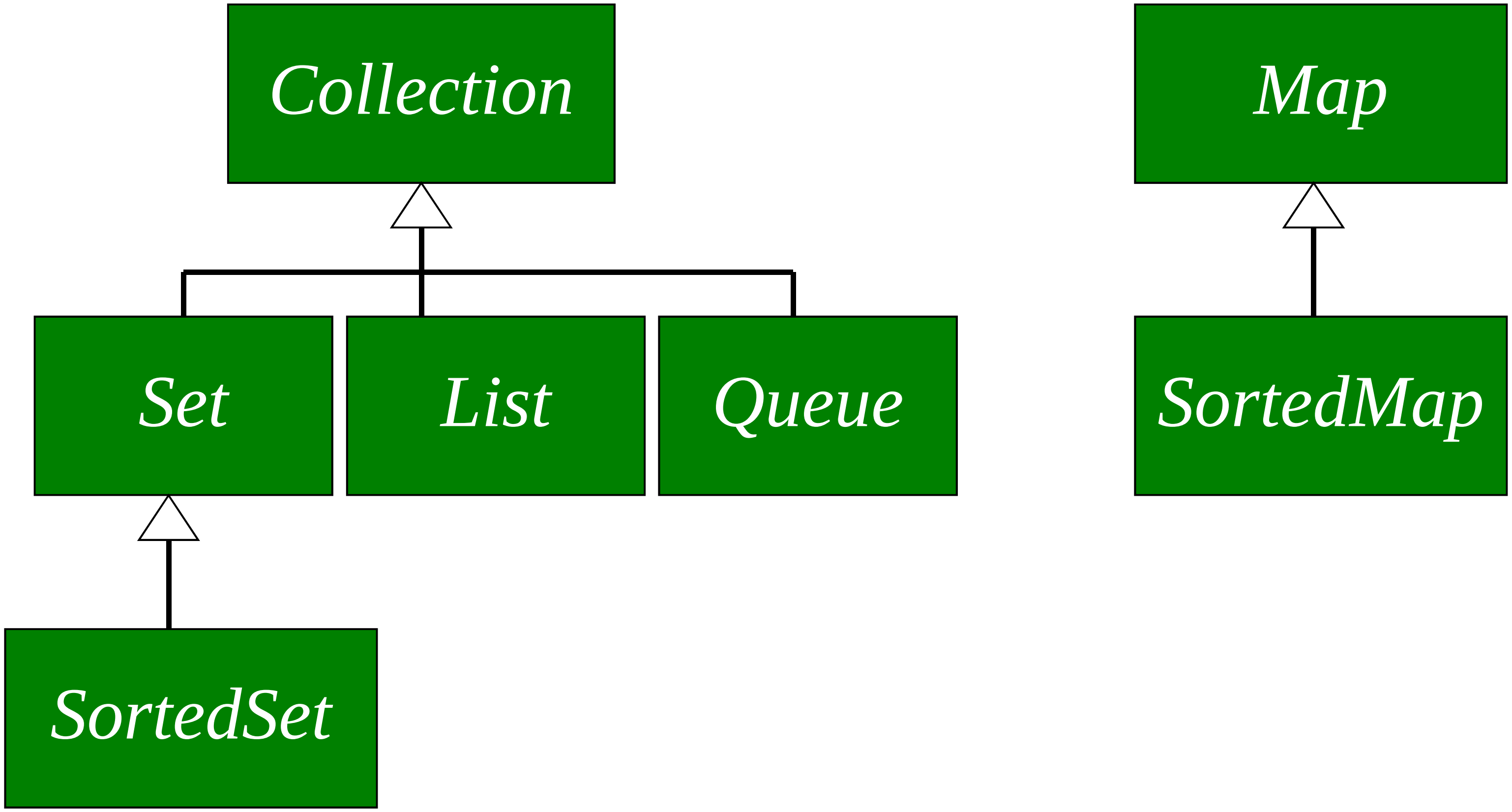
Implementations

- Concrete classes
- Reusable data structures

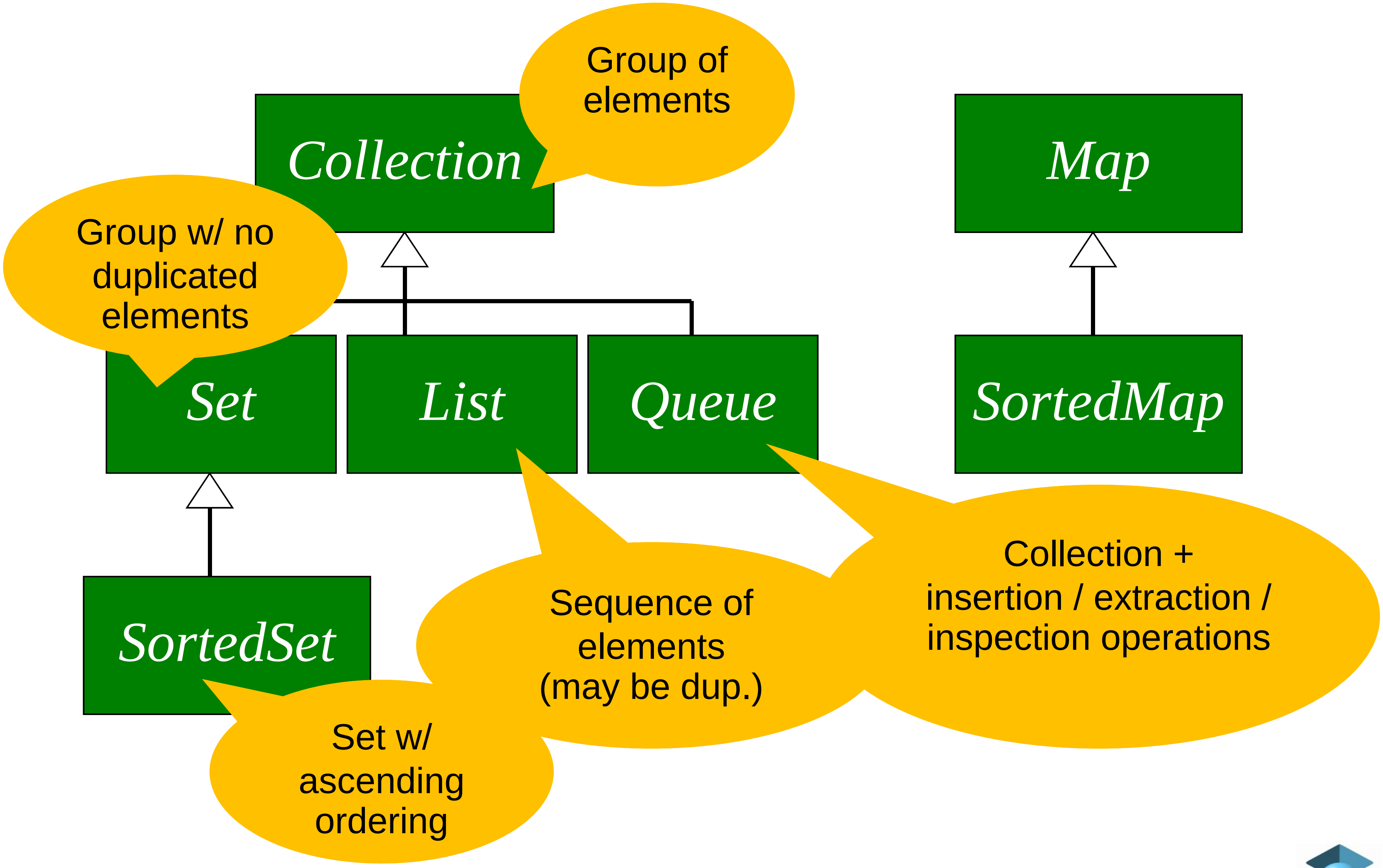
Algorithms

- Useful methods
- Polymorphics → The same method can be used on many implementations.

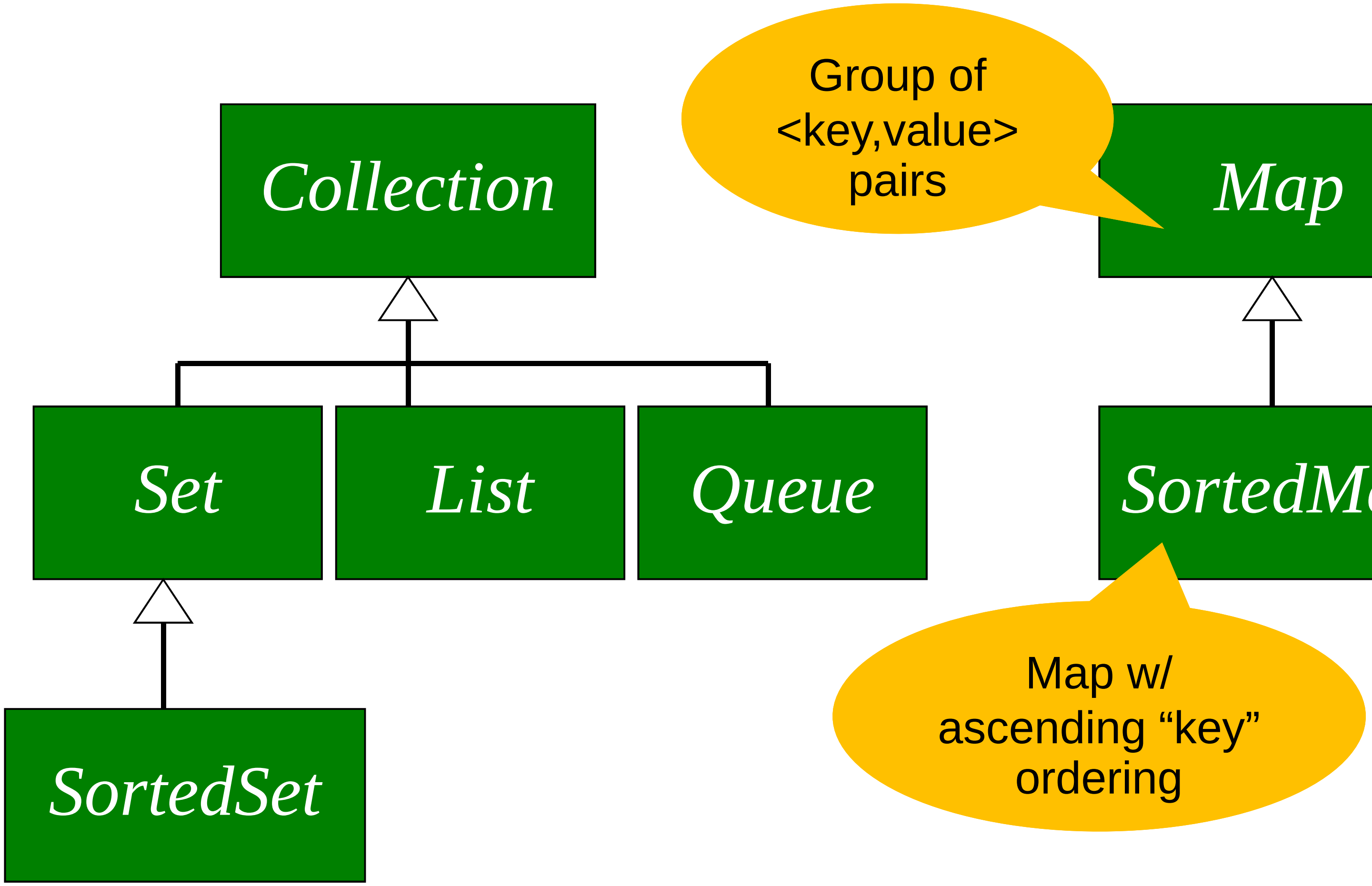
Core Collection Interfaces



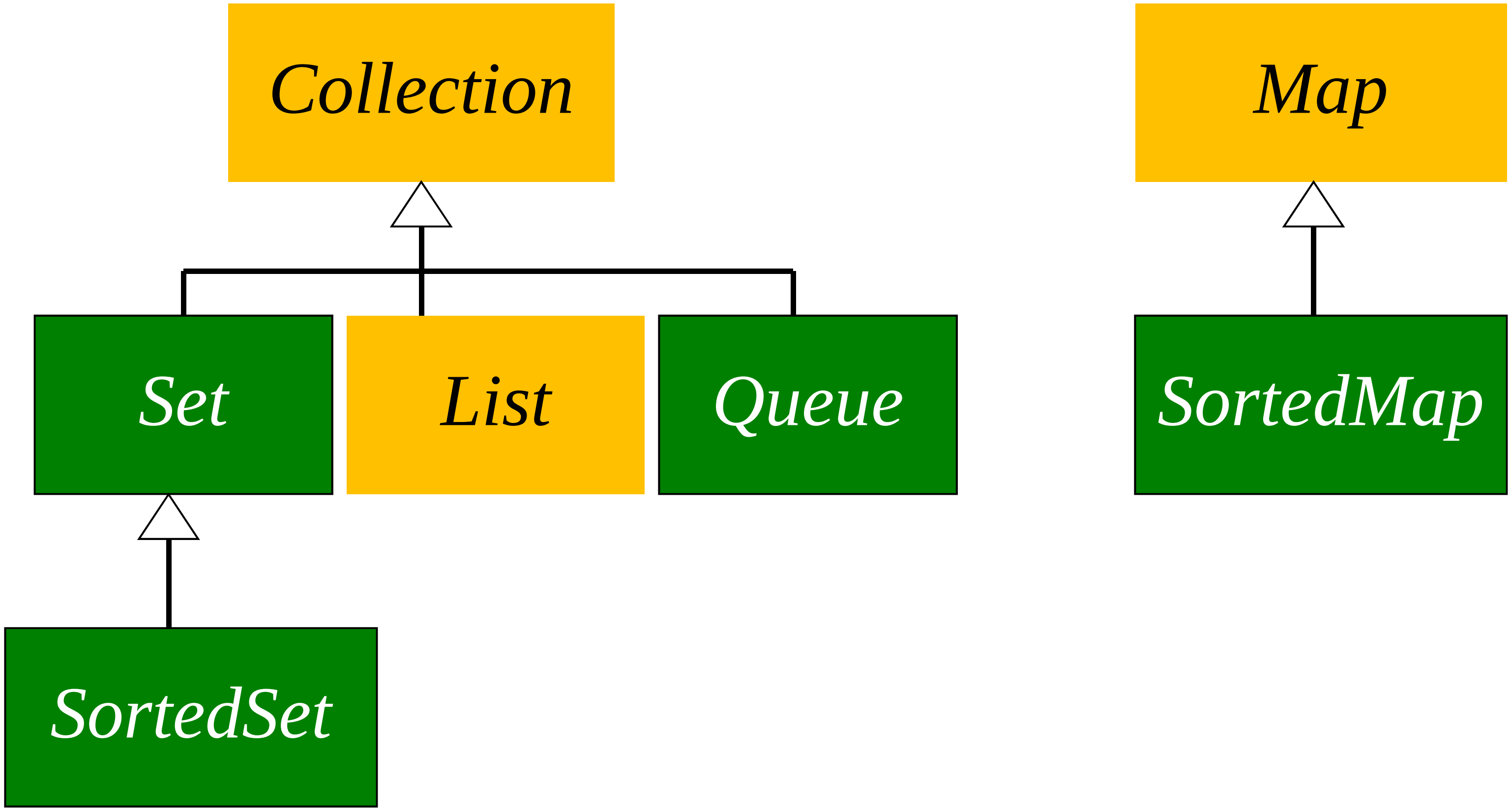
Core Collection Interfaces


Group of elements
Collection
Map
Group w/ no duplicated elements
Set
List
Queue
SortedMap
SortedSet
Sequence of elements (may be dup.)
Collection + insertion / extraction / inspection operations
Set w/ ascending ordering

Core Collection Interfaces



Core Collection Interfaces



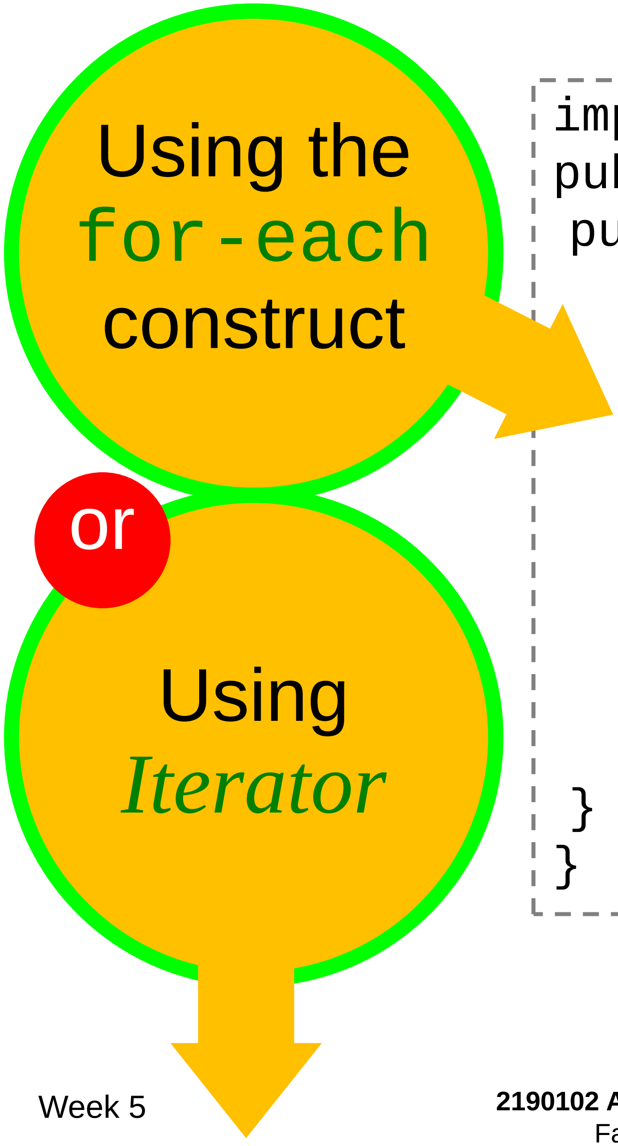
Collection interface

```
public interface Collection <E>
                      extends Iterable <E> {
     //Basic operations
     int size();
     boolean isEmpty();
     boolean contains(Object element);
     boolean add(E element);
     boolean remove(Object element);
     Iterator <E> iterator();

     //Bulk operations
           :
     //Array operations
}
```

Collection interface

```
public interface Collection <E>
                      extends Iterable <E>{
      // Basic operations
                 :
      // Bulk operations
      boolean containsAll(Collection<?> c);
      boolean addAll(Collection<? extends E> c);
      boolean removeAll(Collection<?> c);
      boolean retainAll(Collection<?> c);
      void clear();

      // Array operations
      Object []toArray();
      <T> T []toArray(T [] a);
}
```

Traversing Collections



Using the `for-each` construct

or

Using *Iterator*

```
import java.util.ArrayList;
public class TraversingDemo1{
 public static void main(String []args){
      ArrayList <String> listOfStrings
            =new ArrayList<String> ();
      listOfStrings.add(“ONE”);
      listOfStrings.add(“TWO”);
      listOfStrings.add(“THREE”);
      listOfStrings.add(“FOUR”);
      for(String s :listOfStrings){
            System.out.println(s);
      }
 }
}
```

Iterator

- An iterator is an object that let you:
  - Traversing a collection
  - Removing elements from the collection selectively
- Get an iterator from a collection by calling *iterator()* on that object of collection.

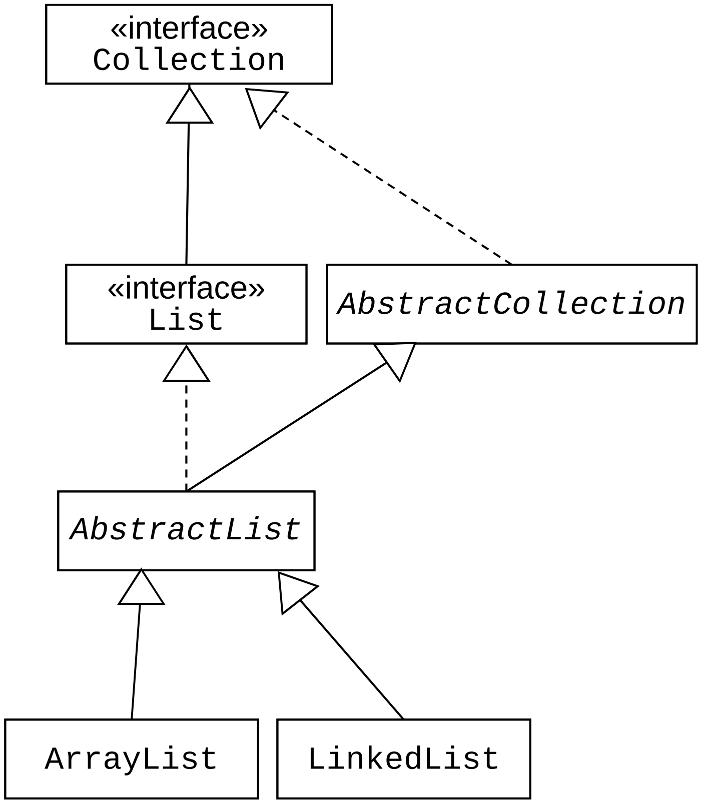
```
public interface Iterator <E>{
    boolean hasNext();
    E next();
    void remove();
}
```

Traversing Collections

Using *Iterator*

```
import java.util.ArrayList;
import java.util.Iterator;
public class TraversingDemo1{
  public static void main(String[] args){
       ArrayList <String> listOfStrings
              = new ArrayList <String>();
       listOfStrings.add("ONE");
       listOfStrings.add("TWO");
       listOfStrings.add("THREE");
       listOfStrings.add("FOUR");
       Iterator <String> it =
              listOfStrings.iterator();
       while(it.hasNext()){
              System.out.println(it.next());
       }
  }
}
```

List : Interface & Implementations



A *List* instance is a

sequence.

The *List* interface contains methods inherited from *Collection* plus:

- Positional Access
- Search
- Range View

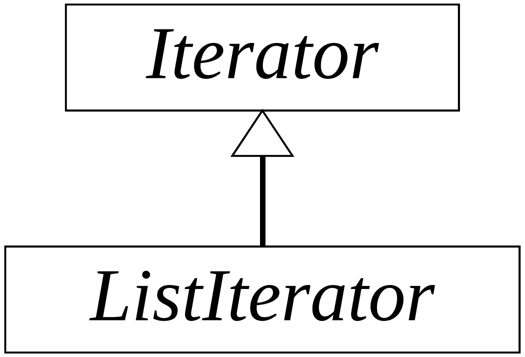
List : Interface & Implementations

```
public interface List <E> extends Collection <E>{
  //Positional access
  E get(int index);
  E set(int index, E element);
  boolean add(E element);
  void add(int index, E element);
 E remove(int index);
  boolean addAll(int index, Collection<? extends E> c);

 //Search
  int indexOf(Object o);
  int lastIndexOf(Object o);

  //Iteration
  //Range-view
}
```

List : Interface & Implementations

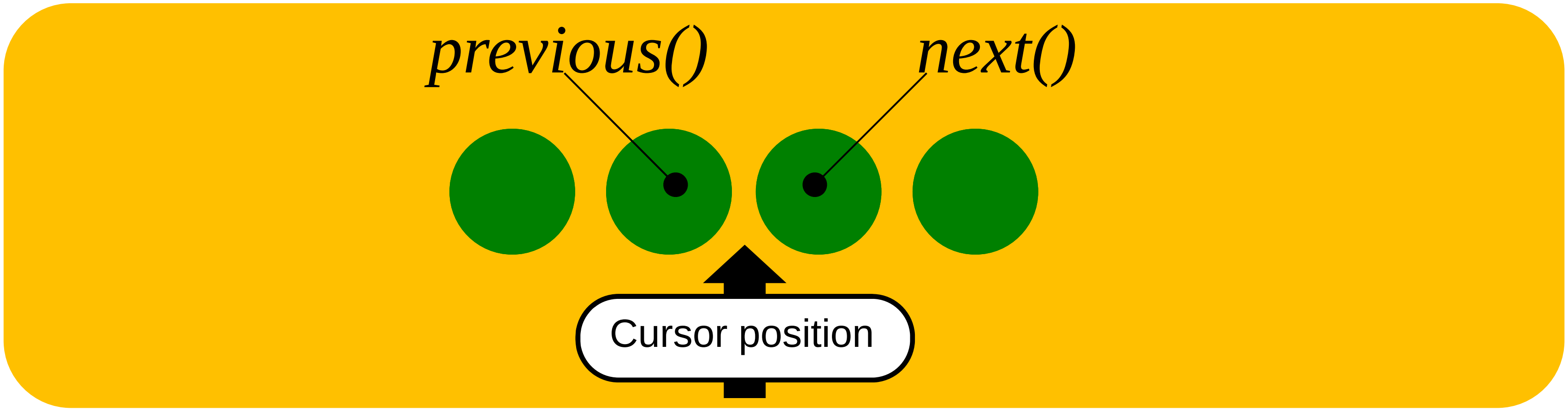
```
public interface List<E> extends Collection<E>{
  //Positional access
        :
 //Search
        :
  //Iteration
  ListIterator<E> listIterator();
  ListIterator<E> listIterator(int index);

  //Range-view
  List<E> subList(int from, int to);
}
```

ListIterator Interface



Allows traversal in either direction.

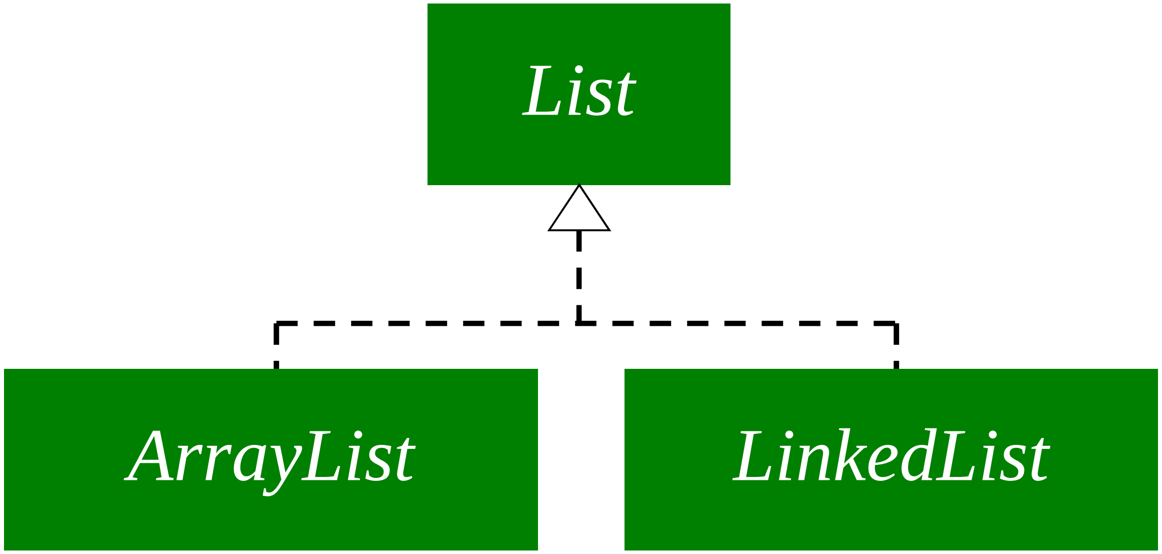
A *ListIterator* has no current element; its *cursor position* always lies between the element that would be returned by a call to *previous()* and the element that would be returned by a call to *next()*



ListIterator Interface

```
public interface ListIterator<E> extends Iterator<E>
{     boolean hasNext();
      E next();
      boolean hasPrevious();
      E previous();
      int nextIndex();
      int previousIndex();
      void remove();
      void set(E e);
      void add(E e);
}
```

List Implementations



Resizable array

- Largely unused space
- Penalty when size exceeds capacity
- More efficient on methods using indices

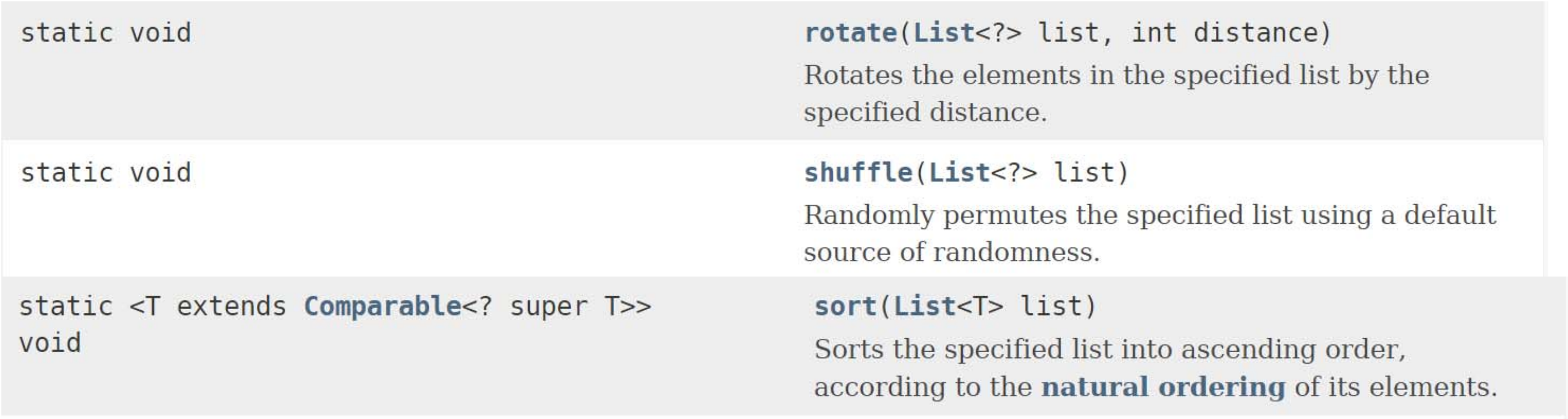
Doubly Linked List

- Waste no space

Polymorphic Algorithms for *List*

java.util.Collections

Static Method Examples:



`static void`	`rotate(List<?> list, int distance)` Rotates the elements in the specified list by the specified distance.
`static void`	`shuffle(List<?> list)` Randomly permutes the specified list using a default source of randomness.
`static <T extends Comparable<? super T>> void`	`sort(List<T> list)` Sorts the specified list into ascending order, according to the **natural ordering** of its elements.

Polymorphic Algorithms for *List*

sort — sorts a *List* using a merge sort algorithm, which provides a fast, stable sort. (A *stable sort* is one that does not reorder equal elements.)

shuffle — randomly permutes the elements in a *List*.

reverse — reverses the order of the elements in a *List*.

rotate — rotates all the elements in a *List* by a specified distance.

swap — swaps the elements at specified positions in a *List*.

replaceAll — replaces all occurrences of one specified value with another.

fill — overwrites every element in a *List* with the specified value.

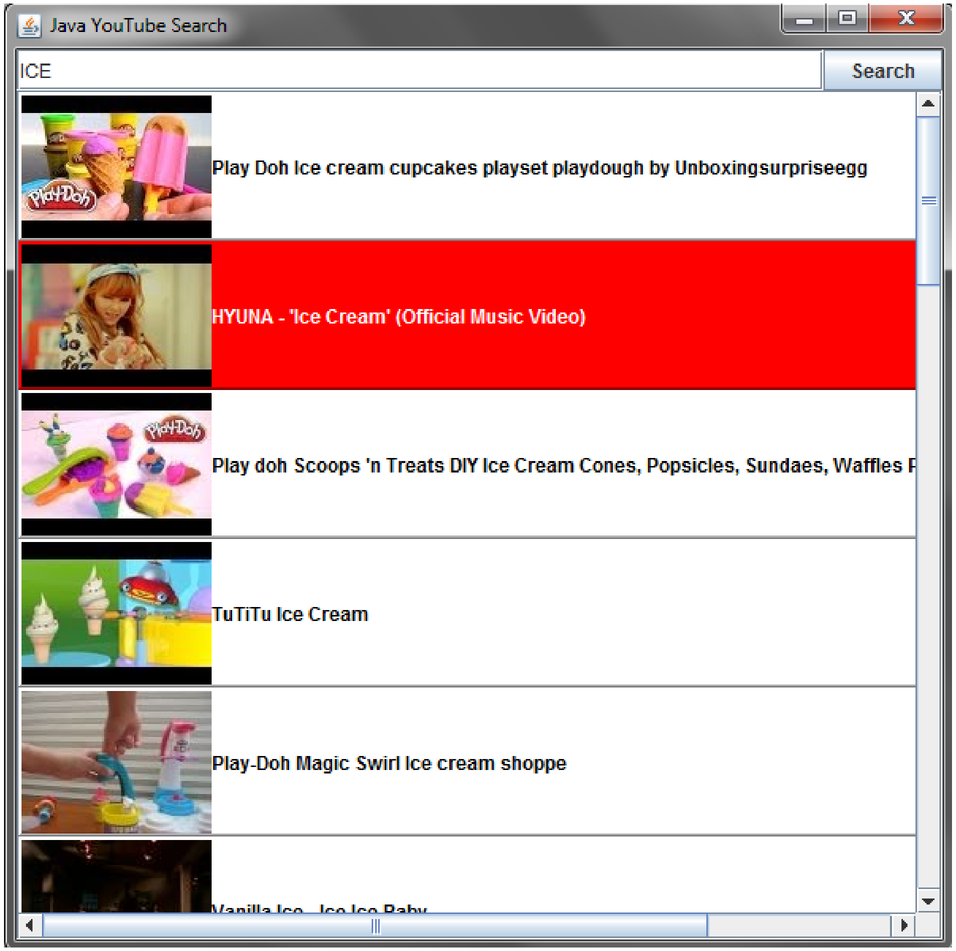
copy — copies the source *List* into the destination *List*.

binarySearch — searches for an element in an ordered *List* using the binary search algorithm.

indexOfSubList — returns the index of the first sublist of one *List* that is equal to another.

lastIndexOfSubList — returns the index of the last sublist of one *List* that is equal to another.

Example



ICE

Play Doh Ice cream cupcakes playset playdough by Unboxingsurpriseegg

HYUNA - 'Ice Cream' (Official Music Video)

Play doh Scoops 'n Treats DIY Ice Cream Cones, Popsicles, Sundaes, Waffles P

TuTiTu Ice Cream

Play-Doh Magic Swirl Ice cream shoppe

Vanilla Ice - Ice Ice Baby

Self-Study for Topic 5

Lesson: Generics (Updated)

http://docs.oracle.com/javase/tutorial/java/generics/index.html

Read all pages, except “Type Erasure” and its subpages.

Trail: Collections

http://docs.oracle.com/javase/tutorial/collections/TOC.html

Read
“The List Interface” & “The Map Interface”

Self-Study Test: Topic 4

The test must be done during:

Saturday 13 September

to Monday 15 September

in the "Assessment" section of

myCourseVille