James Brucker

Collection

A *Collection* is a group of objects.

Set

- an unordered collection
- no duplicates

List

- ordered collection
- duplicates are allowed
- can add or remove elements anywhere in list

Queue and Stack

- like List but obeys a LIFO or FIFO rule.
- can only add or remove from end of collection

Collection Types

Set	List	Stack	Queue
unordered collection without duplicates	ordered collection, may have duplicates	new elements added at top to stack. Only the top element can be viewed or removed.	new elements are added at the "tail" end. Only the top element can be viewed or removed.
people invited to a party	*items in a sale;* *a list of words*	*card game (must remove card from top)*	*customers waiting in line,* *homework to be done.*

Java Interfaces and Classes

Set	List	Stack	Queue
unordered collection without duplicates	ordered collection, may have duplicates	like a List, but can only insert or remove at top	list a List, but can only add or remove at the ends.
Set	*List*	no interface	*Queue, Deque*
HashSet	ArrayList LinkedList	Stack	PriorityQueue

What can you do with a List?

add new elements at the end

add new elements anyplace

get size (how many elements)

remove an element

- remove by name
- remove by position

get some element

find an element (using .equals())

clear the list

copy one list into another list

copy the list into an array

List and ArrayList

List is an interface that defines what "list" can do.

ArrayList is a class that implements List.

List behaves like an *array*, but size can change.

```
ArrayList list = new ArrayList( );
int n = list.size( );  // 0
list.add( "apple" );   // {"apple"}
list.add( "banana" );  // {"apple","banana"}
list.add( 0, "fig" );  // {"fig", "apple","banana"}
n = list.size( );      // 3
Object fruit = list.get( 1 );  // fruit = "apple"
list.remove( 2 );        // remove "banana"
n = list.size( );        // 2
int k = list.indexOf("fig"); // k=0 (index of
"fig")
```

A Type-safe List

To create a List that contains only String.

<String> is called a "type parameter".

```
ArrayList<String> list = new ArrayList<String>( );

list.add( "apple" ); // String
list.add( "banana" );// String
list.add( new Long(10) ); // ERROR! Must be String

String s = list.get(1);   // always returns String
```

Collections are in java.util

```
import java.util.ArrayList;
import java.util.List;

class MyClass {
   private List<String> words;

   public MyClass( ) {
      words = new ArrayList<String>( );
      ...;
   }
```

Why declare "List" instead of "ArrayList"?

We could declare words to be type ArrayList:

```
ArrayList<String> words =
        new ArrayList<String>( );
```

But almost <u>all</u> code declares the variable to be "List". Why?

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

Depend on types, not implementations

Design principle:

"*depend on a specification, not an implementation*",

or: "*program to an interface, not to an implementation.*"

```
class Dictionary {
    private List<String> words =
              new ArrayList<String>( );

   // return the "List" of words
   public List<String> getWords() {
       return words;
   }
```

You have freedom to change the implementation

If your code *depends on an interface* (not concrete class), then you have freedom to change the implementation:

```
class Dictionary {
    private List<String> words =
            // new ArrayList<String>( );
               new LinkedList<String>( );

   public List<String> getWords() {
     // don't let anyone change our word list!
     //   return words;
     return Collections.unmodifiableList(words);
   }
```

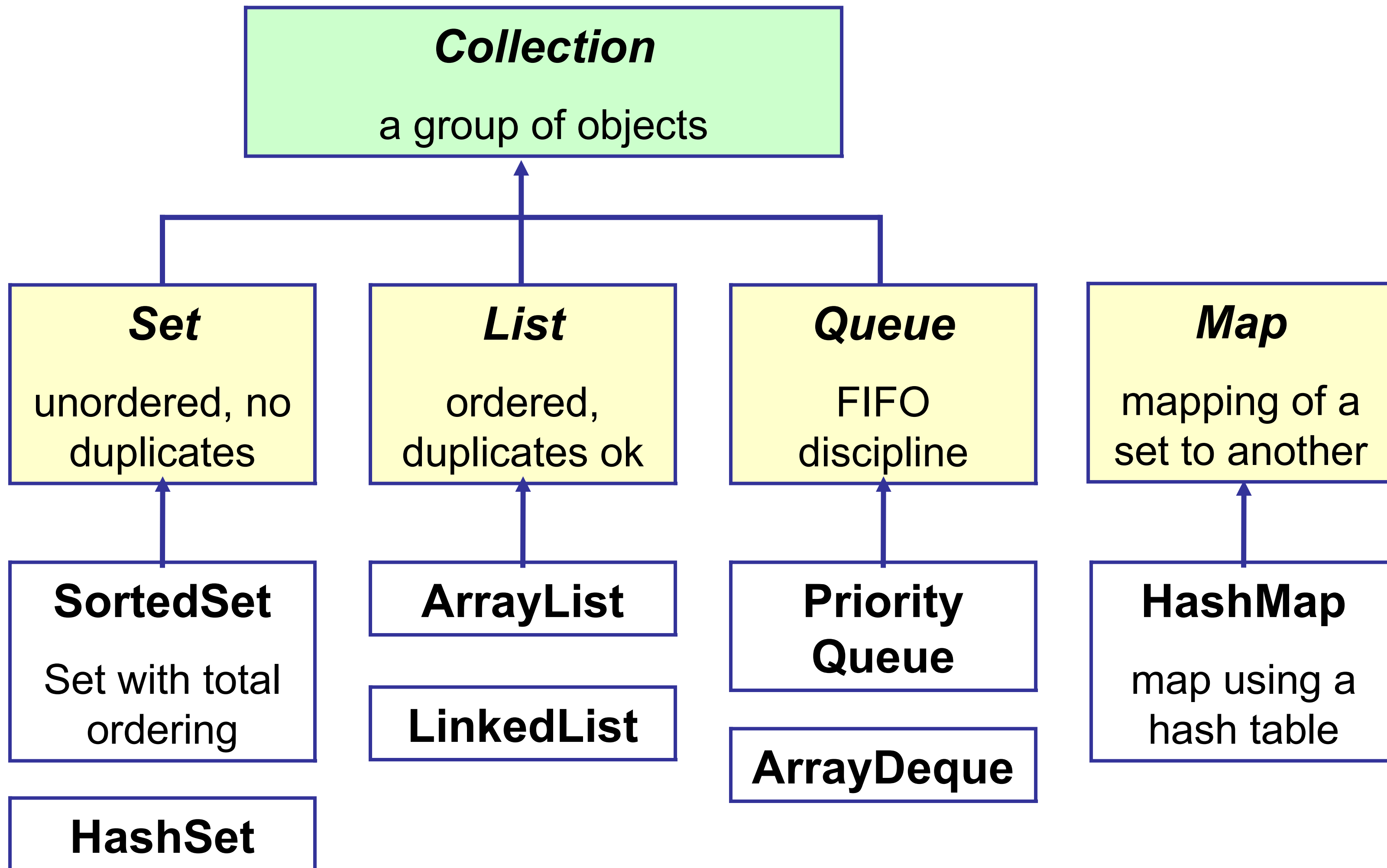
Collections in Programming

Collections are *essential* to many applications.

Example: read words from the console and save them. We don't know how many words.

```
List<String> wordlist = new ArrayList<String>( );
while ( console.hasNext( ) ) {
   String word = console.next( );
   wordlist.add( word );
}
// how many words are there?
int number = wordlist.size( );
// sort the words
Collections.sort( wordlist );
// does list contain the word "dog"
if ( wordlist.contains( "dog" ) ) ...
```

Java Collections



INTERFACES

CLASSES

What can a Collection do?

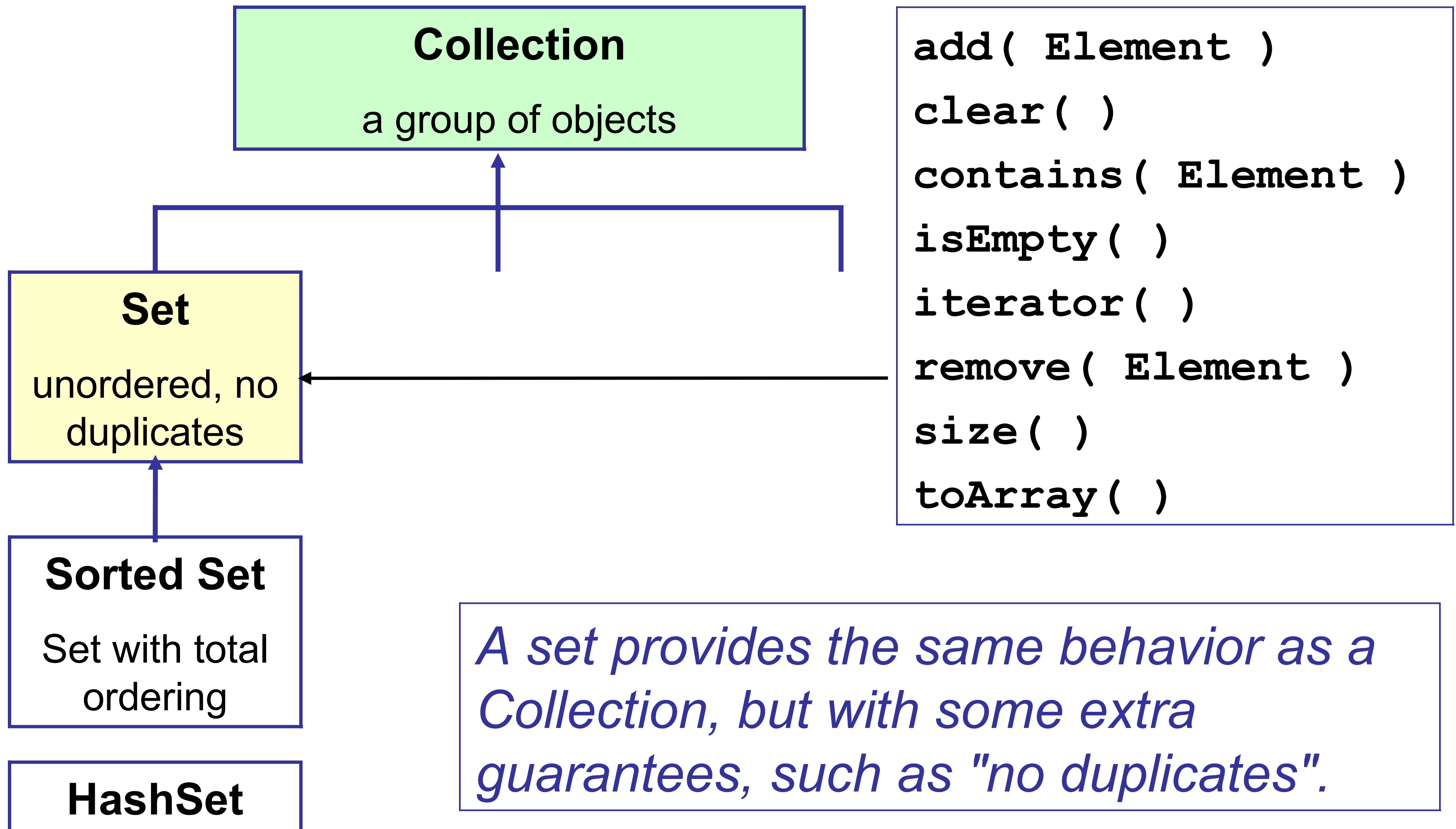
Collection

a group of objects

```
add( Element )
clear( )
contains( Element )
isEmpty( )
iterator( )
remove( Element )
size( )
toArray( )
```

All Collection types have these methods.

What can a Set do?



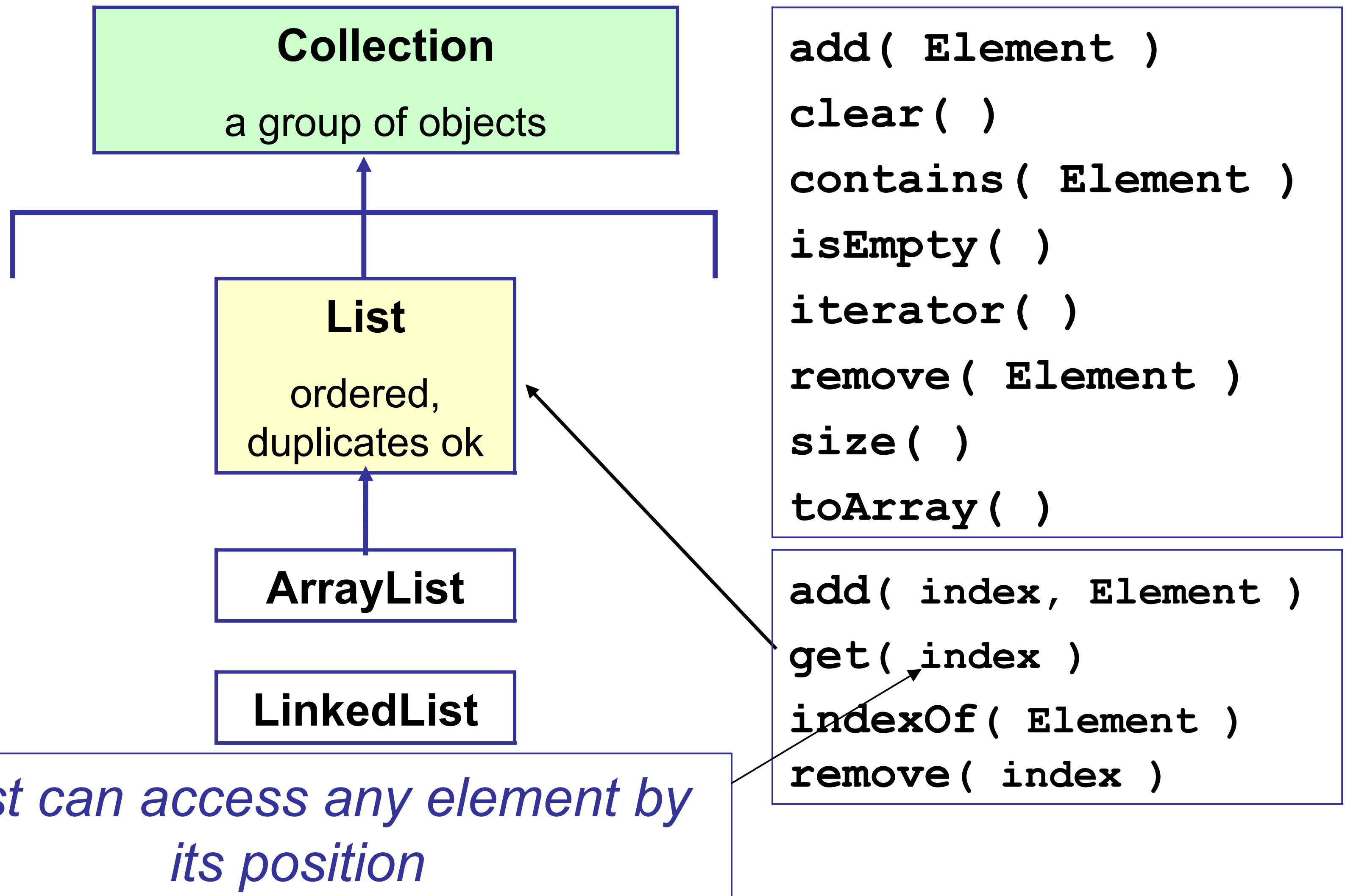
Set Example

Create a set of words.

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


words.add( "apple" );
words.add( "banana" );
words.add( "grape" );
// this should fail
if ( words.add( "banana") ) out.println("added another banana");
else out.println( "couldn't add another banana" );
if ( words.contains("grape") ) out.println( "we have grapes" );
```

What can a List do?



List Example

```
// getClassList returns a List of students
List<Student> myclass;
myclass = Registrar.getClassList("214531");
// sort the students
Collections.sort( myclass );
// print all the students in the class
for(Student st : myclass) {
   System.out.println( st.toString() );
}
```

A "for-each" loop works with any collection

What can a Map do?

Map keys to values

key → value

```
clear( )
containsKey( Key ): boolean
get( Key ) : Value
keySet( ) : get a Set of all keys
put( Key, Value )
remove( Key )
size( )
values( ) : a Collection of values
```

Map

mapping of a set to another

HashMap

map using a hash table

Map: key-value pairs

Map is a "collection" of key-value pairs.
Like a dict in Python.

The Map interface is different from the java.util.Collection interface.

A map of student-id -> Student object

```
Map<Long,Student> map =
                new HashMap<Long,Student>( );
map.add("59101234", new Student("Joe Bruin");
```

Parts of the Collections Framework

Interfaces - define behavior of the collection types

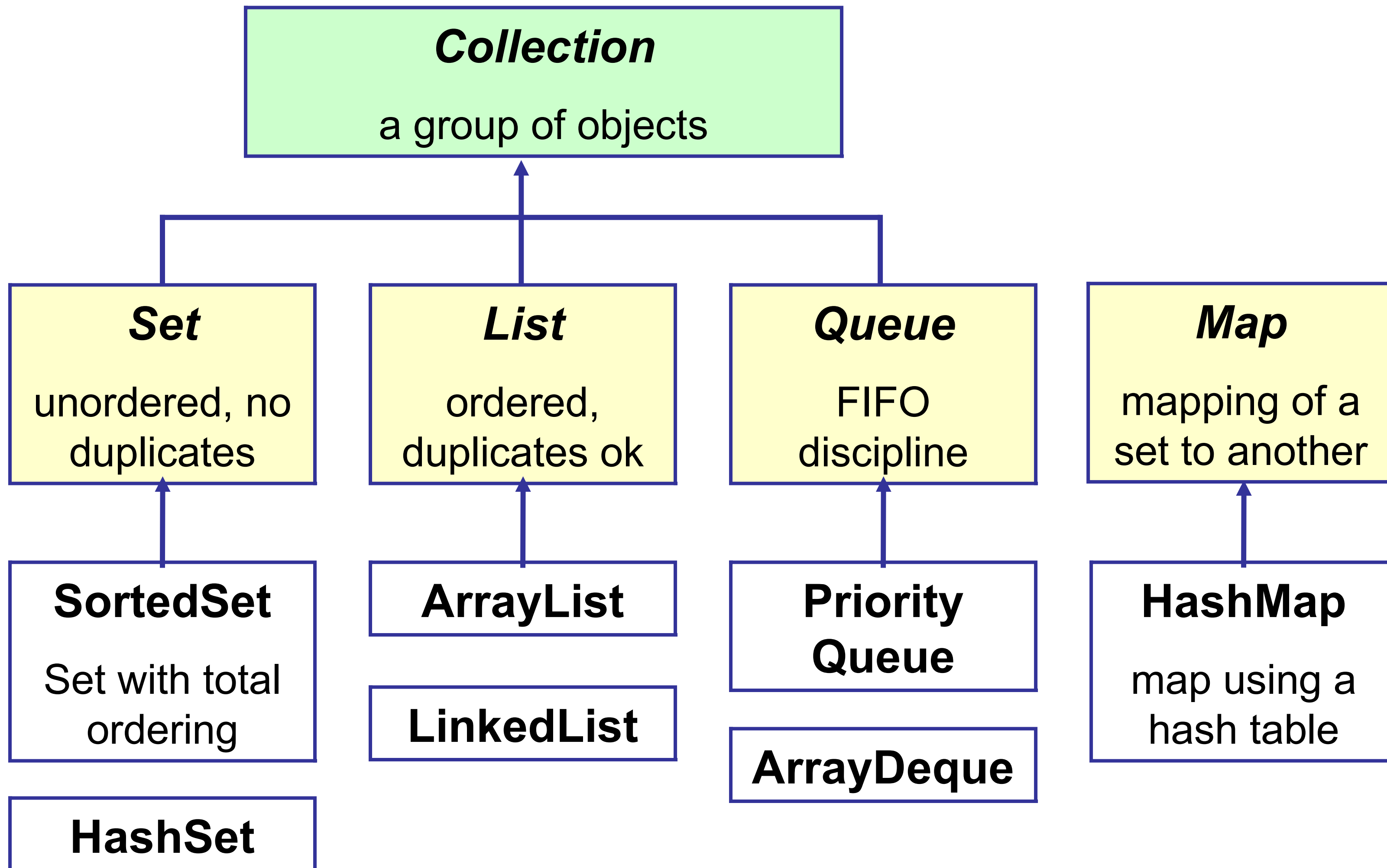
Implementations - abstract & concrete classes

Polymorphic Algorithms - the same operation is implemented by many different types of collection.

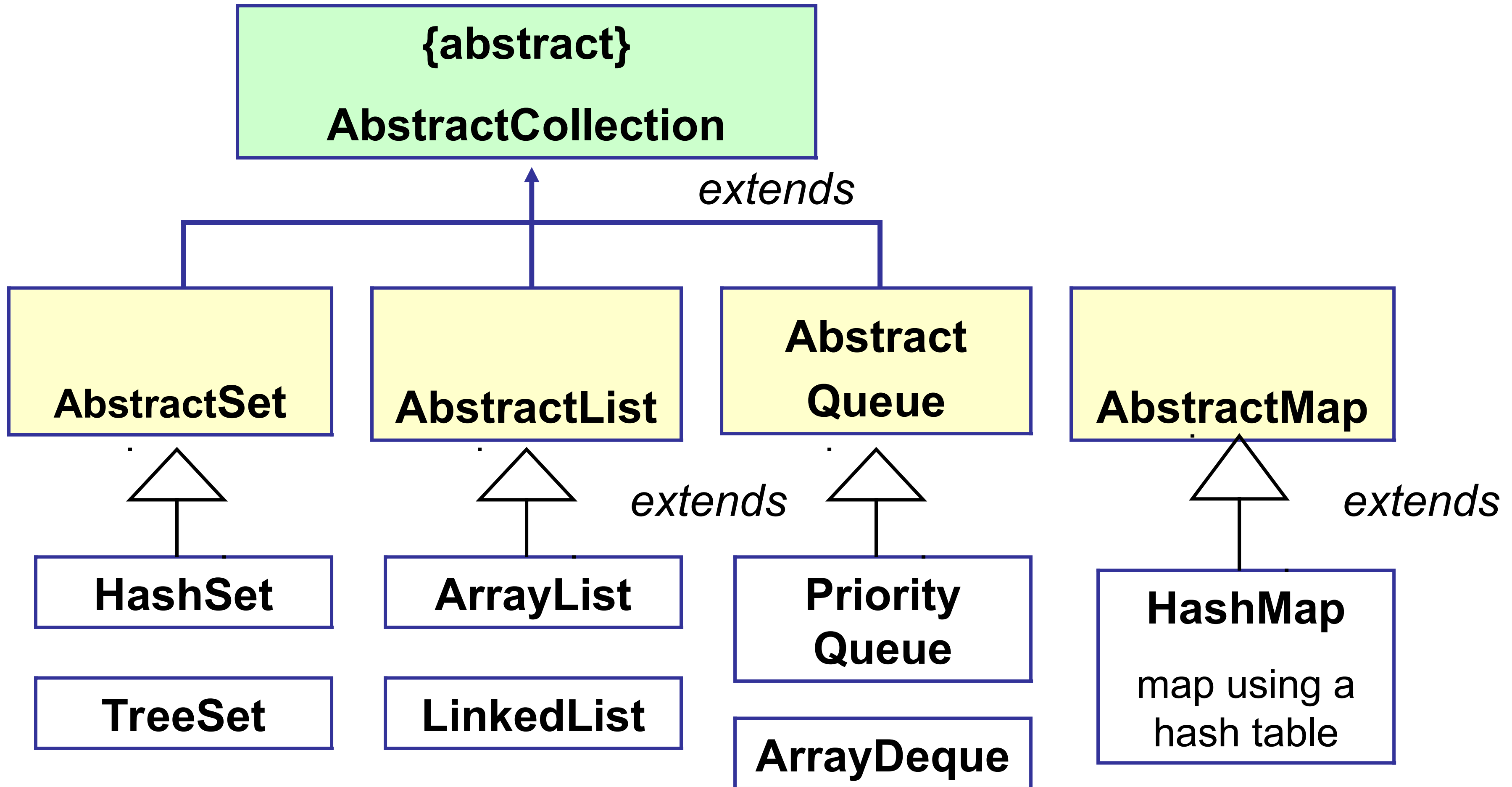
and

the algorithms apply to any data type in the collection.

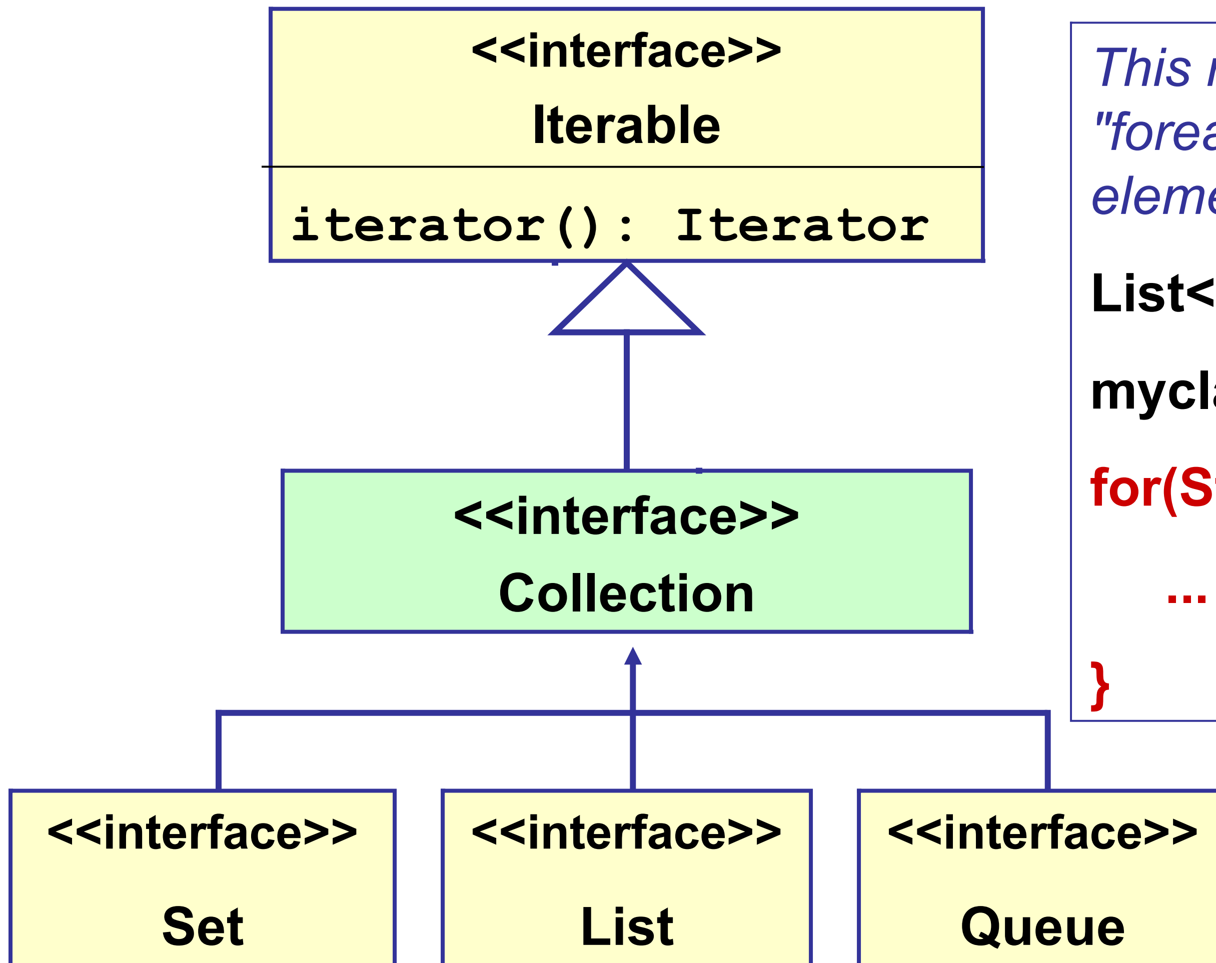
Java Collections



Collection Implementations



All Collections are *Iterable*



This means you can use a "foreach" loop to transverse the elements of any collection.

```
List<Student> myclass;
myclass = regis.getClass( );
for(Student s : myclass ) {
    ...
}
```

What is an Iterator?

Iterator is an interface, with a type parameter **<T>**.

```
interface Iterator<T> {
	// test if there is another element
	boolean hasNext();

	// get the next element
	T next();

	// remove current element (optional)
	void remove();
```

How to Iterate over a Collection

```
Collection collection = new ArrayList() ;
Iterator it = collection.iterator( );
while ( it.hasNext( ) ) {
    Object obj = it.next( );
    // do something with obj
}
```

How to Iterate with type param

```
Collection<String> collection
               = new ArrayList<String>()
Iterator<String> iter
                = collection.iterator( );
while ( iter.hasNext( ) ) {
    String str = iter.next( );
    // do something with str
}
```

Benefit of Iterator

An iterator lets us access elements of a collection without knowing the structure of the collection.

In other words...

*An iterator is an **abstraction** for "iterating" over the elements of a group of things.*

A "for each" loop

```
for-each coin in coinpurse

    add value of coin to the total

end
```

"for-each" syntax

```
for( Type x : Iterable ) { ... }
```

Anything that can create iterator via an iterator( ) method.

```
List<Coin> coins = new ArrayList<Coin>( ) :
for( Coin x : coins ) {
  total += x.getValue( );
}
```

this will call:
`coins.iterator()`

Example of Polymorphism

any collection is OK

```
Collection<Student> coll = new LinkedList<Student>( );
coll.add( new Student( id1, name1) );
coll.add( new Student( id2, name3) );
int n = coll.size( );
// Sort the students
Collections.sort(coll);
// Display the sorted students
for(Student student : coll)
    System.out.println( student );
```

Array Operations

- Every Collection has a **`toArray()`** method.
- This lets you copy **any collection** into an array.
- The array contains ***references*** (not copies) of the objects in the collection.

```
Collection<Student> myclass =
                    Registrar.getClass( "219141" );
// create an array to hold the students
Student[ ] array = new Student[myclass.size( )];
// copy the List to array
myclass.toArray( array );
```

Resource

- "*Collections Trail*" in the *Java Tutorial*
  http://java.sun.com/docs/books/tutorial/index.html