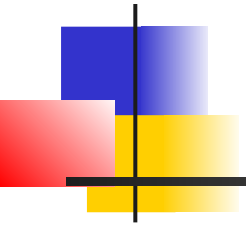


Collections

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/remove at top	list a List, but can only add 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")
```

Problem with "List"

A basic List can contain *any* type of object!

```
ArrayList list = new ArrayList( );
list.add( "apple" ); // String
list.add( new Date( ) );  // Date object
list.add( new Long(10) ); // Long object
list.add( new Coin(5) );  // Coin

Object obj = list.get(1); // always returns object
```

list.get( ) requires a *cast*

Since List (or ArrayList) can contain any kind of Object, we have to ***cast*** the result to what we want. This is messy code and can cause errors.

```
ArrayList list = new ArrayList( );

list.add( "apple" ); // String

list.add( "banana" );

...

String x = (String)list.get(1); // cast to String
```

A Type-safe List

To create a List that contains only String.

```
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
```

List and ArrayList 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 all code declares the variable to be "List". Why?

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

Depend on types, not concrete implementations

A design principle "*depend on abstraction, not on concretion*", 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 implemenation:

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

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

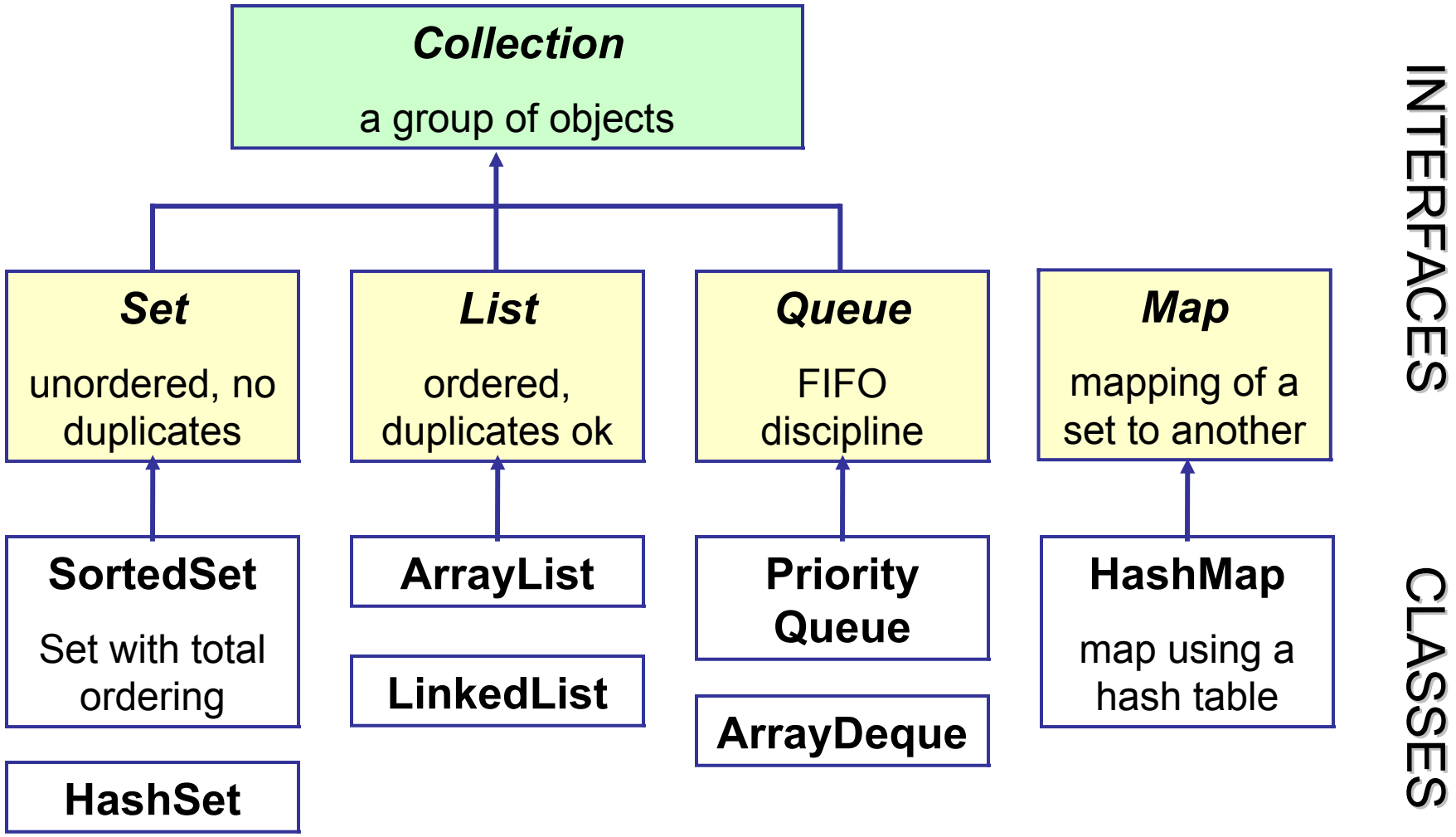
Collections in Programming

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



What can a Collection do?

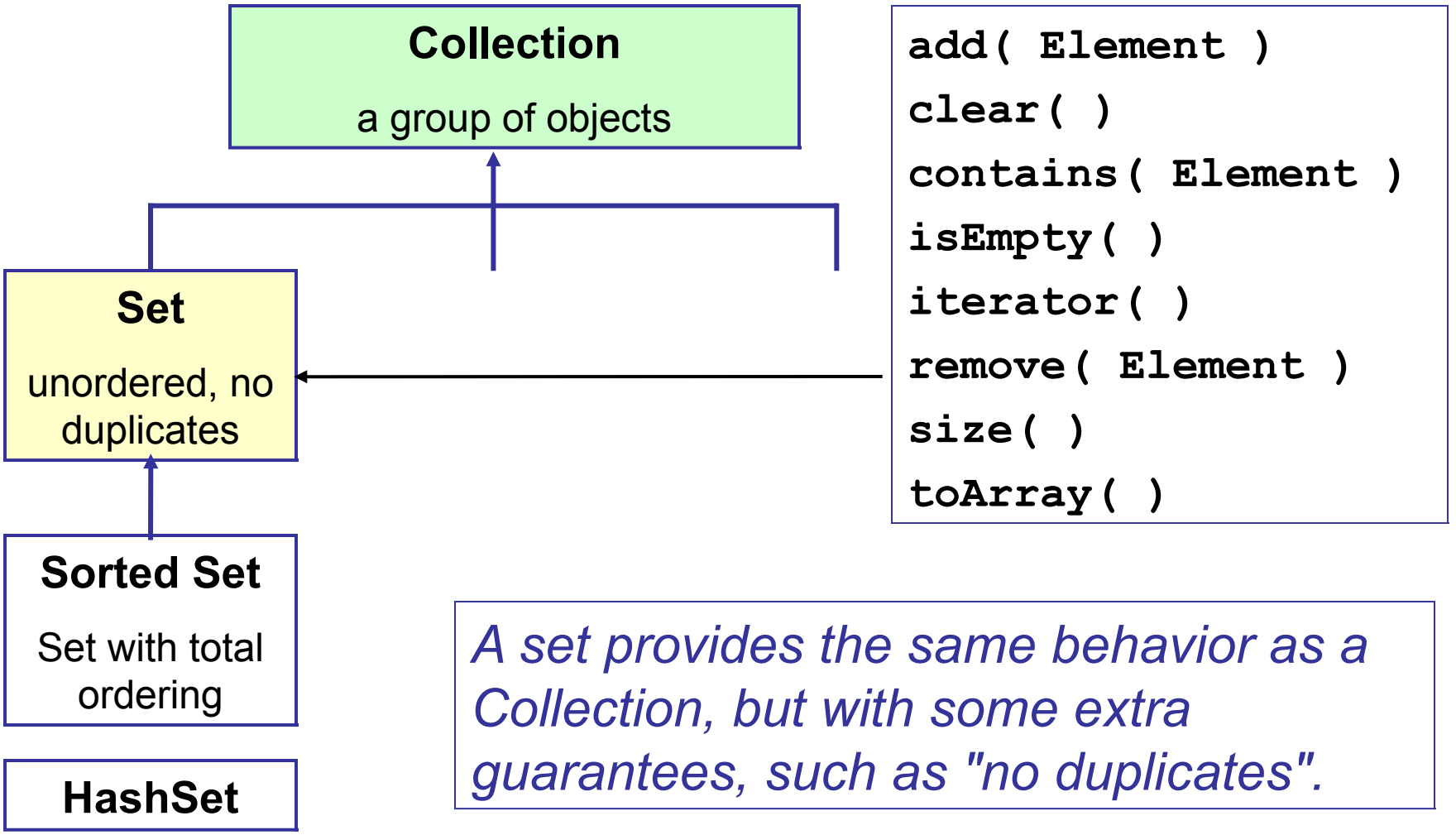
Collection

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?



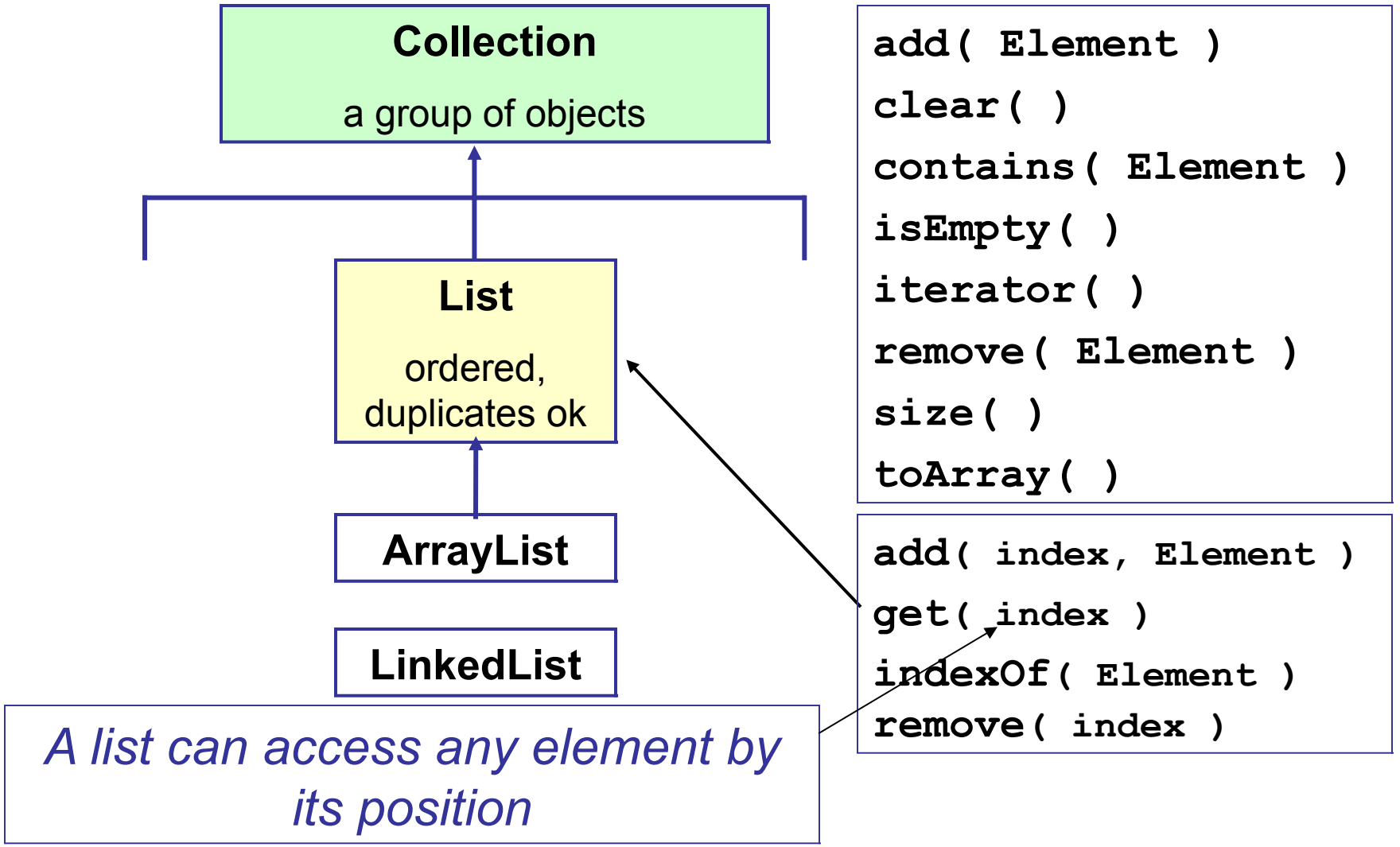
A set provides the same behavior as a Collection, but with some extra guarantees, such as "no duplicates".

Set Example

Create a set of words.

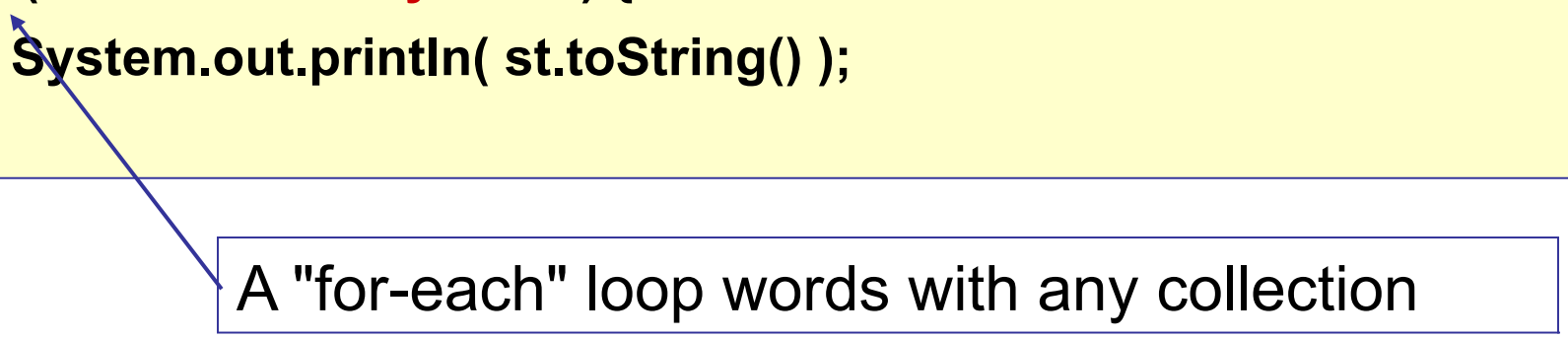
```
Set words = new HashSet( );


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

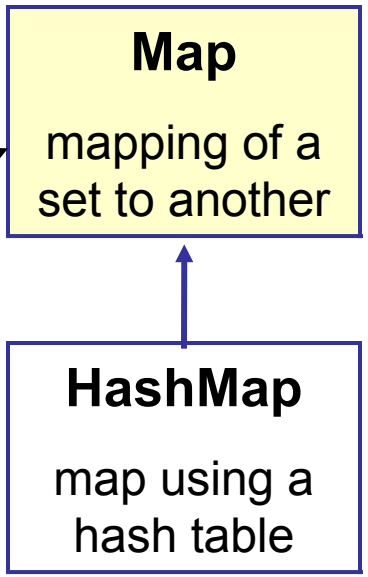
```
// getClassList returns a List of students
List<Student> myclass  = new ArrayList<Student>( );
myclass = Registrar.getClassList( "214531" );
// sort the students
Collections.sort( myclass );
// print all the students in the class
for ( Student st : myclass ) {
    System.out.println( st.toString()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



What can a Map do?

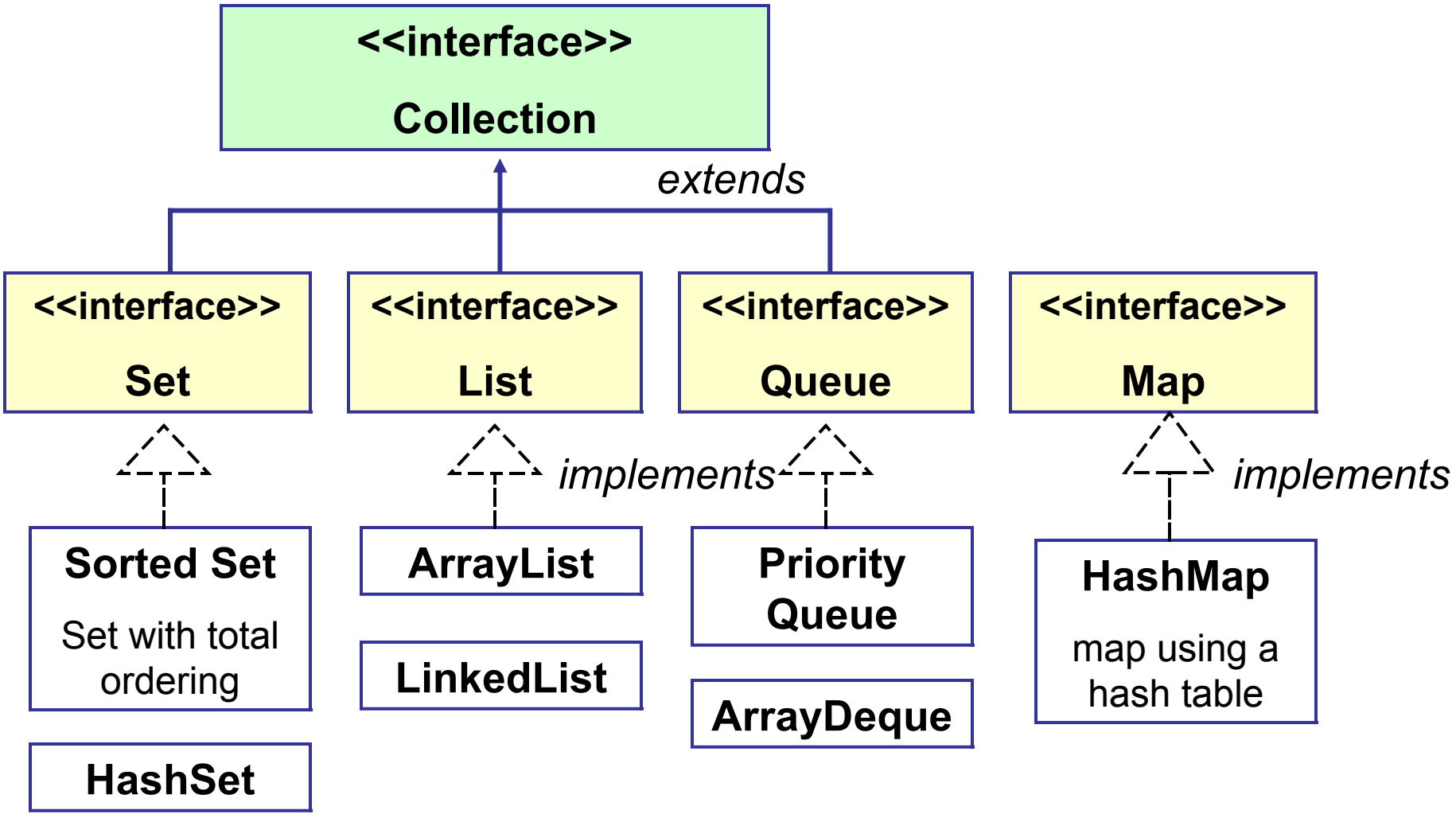
Map keys to values

key → value

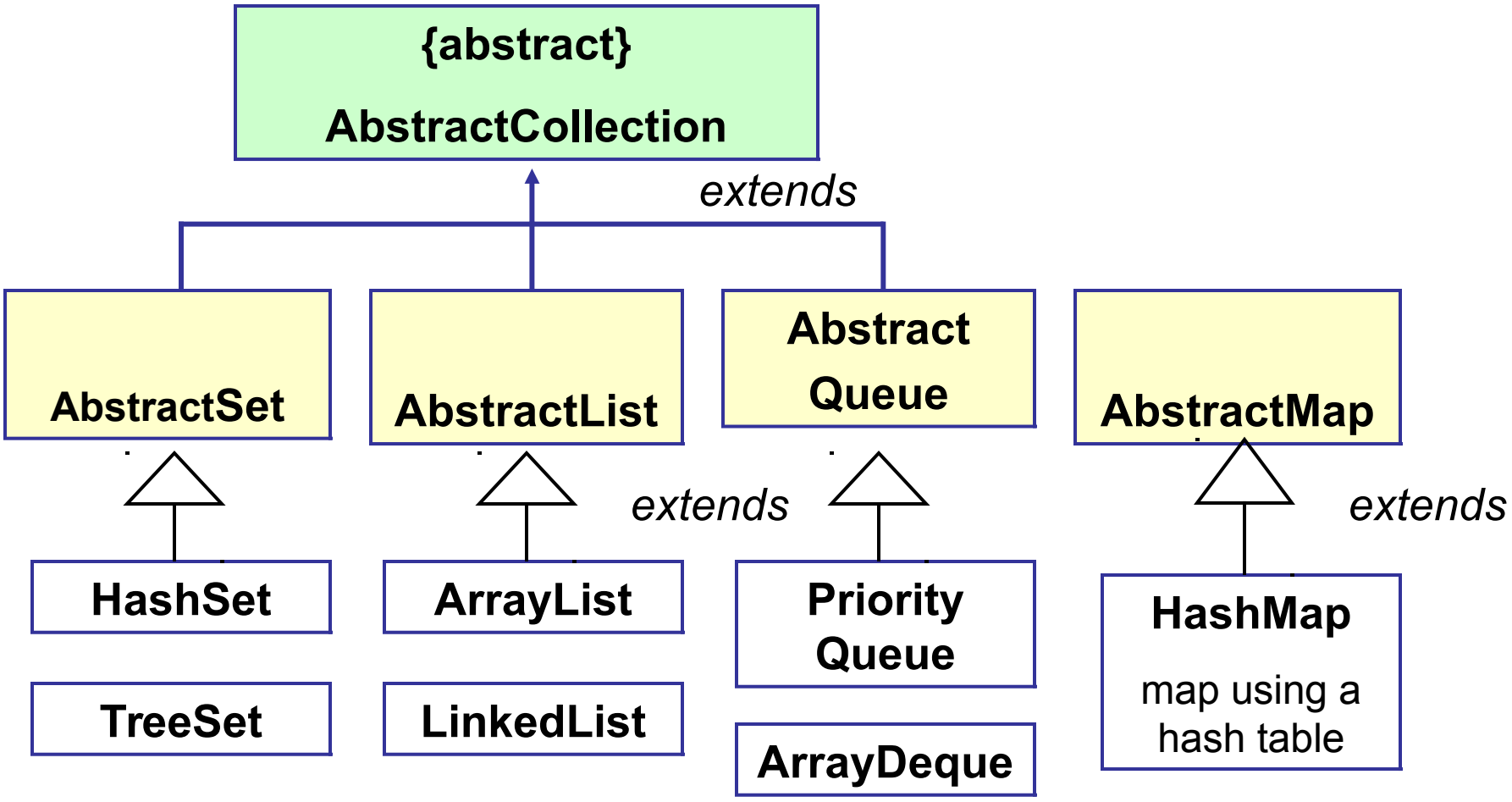



```
clear( )
containsKey( Key ): boolean
get( Key ) : Value
keySet( ) : get a Set of all keys
put( Key, Value )
remove( Key )
size( )
values( ) : a Collection of values
```

The Java Collections Framework



Collection Implementations



Collections without Type parameter

Collection of any kind of object:

```
List myclass = new ArrayList( );
// you can put anything in the collection
myclass.add( new Student( "Nerd" ) );
myclass.add( new Date() );
myclass.add( new Double(2.5) );
// "get" method returns Object
Object obj = myclass.get( k );
```

Collections with a Type parameter

Use a *type parameter* in <. . .> to restrict elements:

```
List<Student> myclass =
    new ArrayList<Student>( );
// you can put only Student in myclass
myclass.add( new Student( "Nerd" ) );
myclass.add( new Date() );  // ERROR
// "get" method returns Student
Student st = myclass.get( k );
```

Collections using <Type> parameter

Type param is for **safety** and **easier** programming.

You don't need to cast to convert to data type.

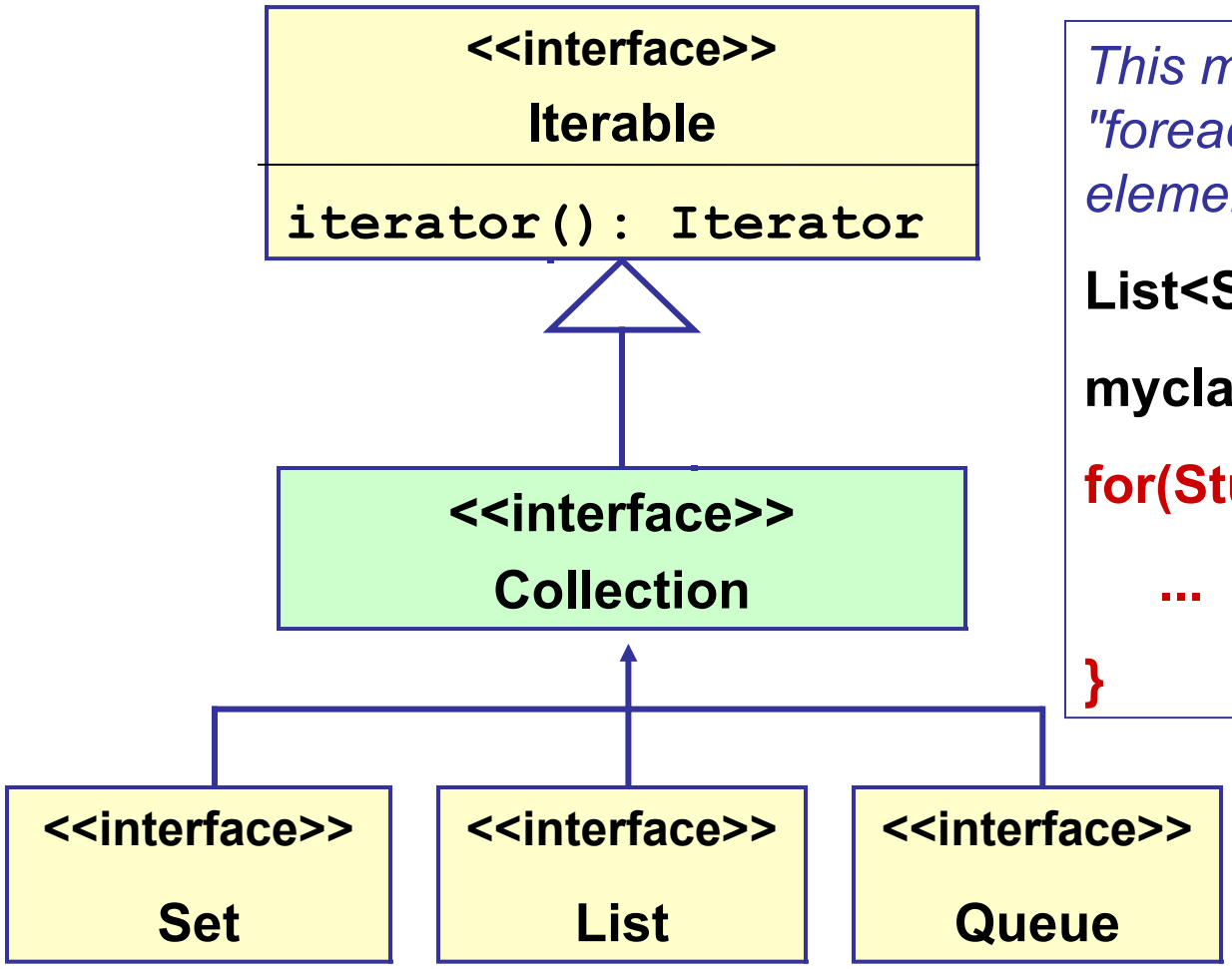
```
Student student = (Student)list.get(k);
```

```
List<Student> myclass = new ArrayList<Student>( );
Student who = myclass.get( 0 );
```

myclass can only contain elements of type "Student".

myclass.get(k) always returns a Student object.

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 ) {

    ...

}
```

How to Iterate over a Collection

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

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 a collection of things.

A "for each" loop

```
for-each x in myAddressbook
    call x and invite him to the party
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()`

Parts of the Collections Framework

Interfaces - define behavior of the collection types

Implementations - the concrete collection 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.

Example of Polymorphism

any implementation 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  );
```

Exercise

- For the KU Coupons application, write a Store class that records all the Transactions in an ArrayList.
- Write a getTotalSales( ) method that sums the sales amount in all Transactions.
- Write a printSalesReport( ) to print the sales.

Map: not a Collection

Map
collection of key-value pairs. Has methods to find a value given the key.
Students, indexed by name or student id. *Products in a Store, indexed by product id (barcode).*

```
Map<Long,Student> map =
                    new HashMap<Long,Student>( );
map.add("111", new Student("Joe Bruin");
```

Map Example

Create a map between Coupon colors and money values.

```
Map<String,Integer> map = new HashMap<String,Integer>( );
map.put( "red", 100 );  // red is worth 100 Baht
map.put( "blue", 50 );  // blue is worth 50 Baht
map.put( "green", 10 ); // green is worth 10 Baht
int value = map.get( "blue");   // returns value = 50
// now process a coupon
String color = console.next( );  // get the coupon's color
if ( map.containsKey( color ) ) {
    value = map.get( color );
    total = total + value;
}
```

Resource

- "*Collections Trail*" in the *Java Tutorial*

  http://java.sun.com/docs/books/tutorial/index.html