

# Collections

(the old style – pre 1.5)

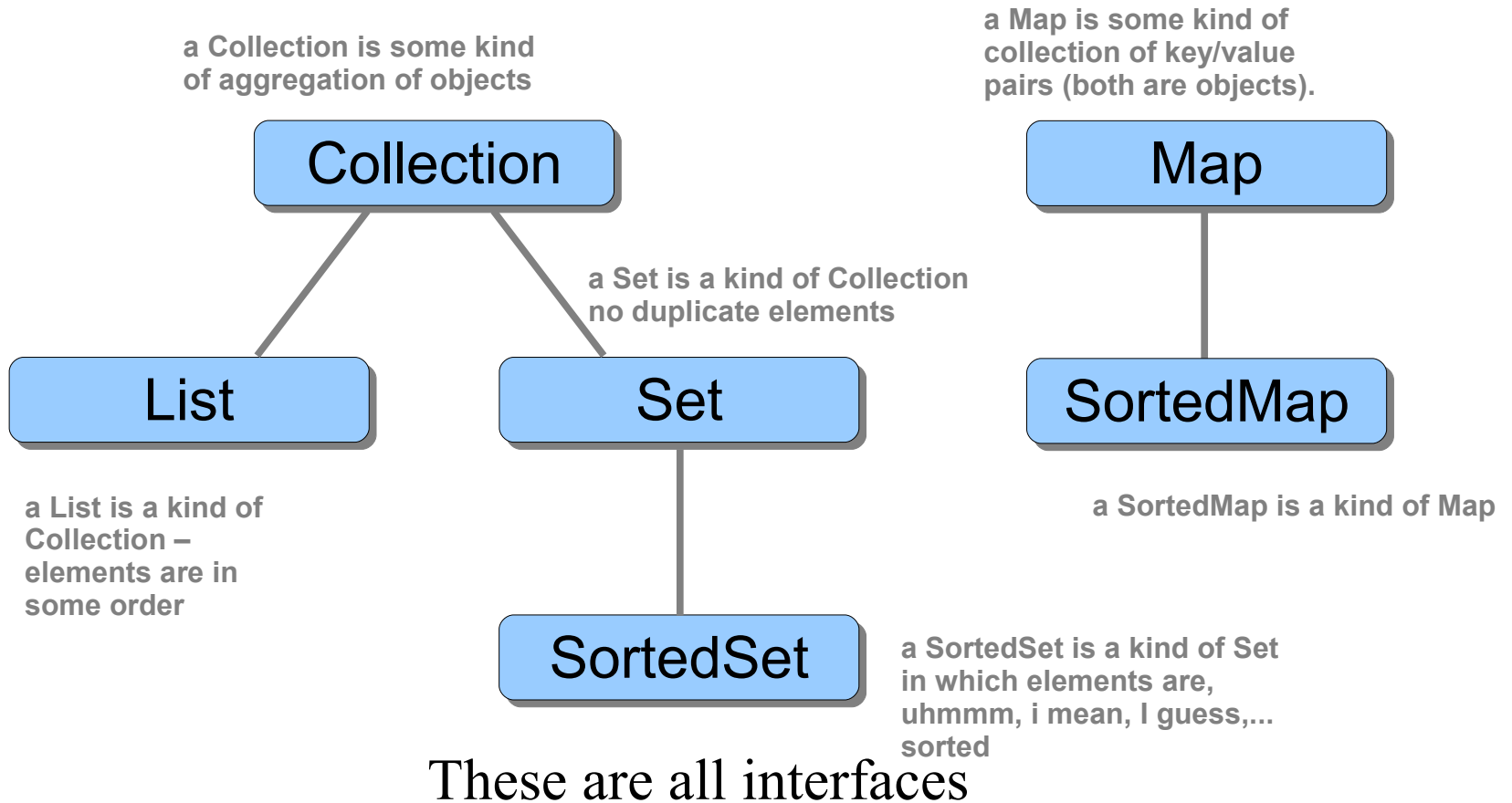
# java.util Collections

- We start by looking at the *old style* collection support (before JDK 1.5).
  - JDK 1.5 adds *generics* (like C++ templates).
- The basic idea is similar to STL:
  - provide a set of generic data structures that can hold lots of different kinds of data.
  - provide a common programming interface to these data structures.
  - provide some *generic* methods that will run on these data structures (sorting, searching, etc).

# Compatibility Issues

- JDK 1.5 supports the old style collections, but you see some compiler warnings.
- You will probably want to use the new style!
  - less typing (type casting).
- In general, the new style changes how you declare data structures, but not how you use them...

# Simplified Collections Hierarchy



# Some Collection methods

If you have any kind of List or Set (or anything else that is a Collection), you can do this kind of stuff:

- `add(Object)` adds an object to the collection.
- `remove(Object)` remove an object.
- `contains(Object)` search for an object.
- `size()` find out how many objects there are.
- `toArray()` create an array of object references from the collection.
- there are lots more...

# Some List methods

If your Collection is a List, you can also get elements according to their position in the list:

- `add(int, Object)` add (insert) new object at specific position.
- `remove(int)` remove according to position in list.
- `get(int)` get an object given it's position in the list.
- `set(int, Object)` change what object is at some position

# Set methods

Set doesn't add any methods to those specified in `Collection`.

You can't add an object that is already in the Set

- `add()` returns false.

`SortedSet` adds methods

- `subSet()`, `first()`, `last()`, `comparator()` and more...

# Iterators

- An iterator is an object than can step through the elements in a collection.
  - any kind of collection!
- You ask the Collection to create an iterator for you:

```
Iterator i = c.iterator();
```

# Iterator Methods

- Just two:
  - Object `next()` returns next object in the collection.
  - boolean `hasNext()` is there any more?
- This allows you to write a method that could operate on *any Collection*.

# Iterator Example

```
void printCollection(Collection c) {  
    Iterator i = c.iterator();  
    while (i.hasNext()) {  
        System.out.println(i.next());  
    }  
}
```

# So what...

- Collection, List, Set and SortedSet are all interfaces.
  - we can't create any of these directly, we can only create an object that implements these interfaces.
- We need Classes/Objects that actually implement these interfaces.
  - there are a variety of these, in general you need to understand how they support the interfaces to pick the best one for your application.

# Things that implement List

- There two (primary) classes that implement List:
  - ArrayList stores the elements in an Array.
    - provides fast access to any element given an index.
    - Not so fast insertion and deletion.
  - LinkedList take a wild guess how this stores the list...
    - slower access given an index, but much faster insertion and deletion.

# Sample Code

- `ArrayListPlay` populates an `ArrayList` with random numbers and prints them out.
- `ArrayListSort` Uses `Collection.sort` and a comparator object.
- `ListPrinting` print an entire list with one call to `System.out.println`

# Using LinkedList

- Just say "LinkedList" instead of "ArrayList".
  - You don't need to change any code.
  - Depending on the application, the performance will change.
- Keep in mind: eventually you will come across situations in which you write code that deals with a List (someone else gives you a List and you don't ever know what kind!).

# Things the implement Set

- HashSet uses a has table to access individual elements quickly.
  - contains ( ) method is faster than with a list.
- TreeSet is actually a SortedSet
  - iterator will retrieve elements in order
  - the order is user-definable using comparators.

# A Set of samples

- `HastSetPlay` put a bunch of random numbers in a set (watching for duplicates).
- `TreeSetPlay` put a bunch of random numbers in a sorted set, then print out in order.

# Maps

- A map is a collection of key/value pairs.
  - given a key, we want to be able to lookup the value.
- For example: name/grade pairs:

bode , 23

joe , 100

paris , 22

sam , 95

sue , 98

Why not just a Collection of objects, each of which has a name and a grade?

We want to be able to get a grade given a name. We don't want to search through the list ourselves...

# Map interface

- Both the key and the value are objects.

```
put(Object key, Object value)
```

```
get(Object key)
```

```
remove(Object key)
```

```
size()
```

there is more...

# Things that implement Map

- `HashMap` is the basic implementation.
  - there are others: `Hashtable`, `LinkedHashMap`
- `TreeMap` implements `SortedMap` (which is a kind of `Map`...)

# Map sample code

- **HashMapPlay** Does word to number conversions using a map.
  - you type in 'one', it prints 1. 'eight' -> 8., etc.
  - allows you to type "three plus five" and computes the sum.
- **CountWords** word frequency counting program.
  - compares performance using a HashMap and a TreeMap.