Collections

- Collections
  - Lists (continued)
  - Maps
  - Sets
- Iterators
- Comparators

Caveat

- Today's lecture is to give you some examples and background knowledge of what's available in Java
- Don't worry about remembering specific class and method names or code examples (Those are easy to look up!)
- The important thing to take away is what is available to you as a programmer.
- If you know what the language is capable of, it's easy to look up how to use it

Collections

- Java provides a library called the *Java Collections Framework*.
- The Collections framework is a library of classes for working with *sets* of objects
- A collection is an object which can store other objects
- An **ArrayList** is a type of collections

Collections

- Collections store a set of *elements*
- To store elements there must be methods for:
  - Adding elements to the collection
  - Removing elements
  - Retrieve elements that have been added to the collection

Collections

- There are three main types of collection:
  - Lists
  - Sets
  - Maps

Lists

- Lists store elements sequentially based on an integer index.
- The index is automatically calculated inside the list object (The programmer doesn't specify the index when adding an element) e.g. list.add(“A”)
- Lists permit duplicate elements (an element can exist in more than one position in the list)
- The **ArrayList** is a type of list

Sets

- Sets have no notion of position or order
- Sets do not permit duplicate elements
- Adding the same element to the set multiple times will have no effect
- To retrieve an element from a set you must just iterate over it, you cannot just retrieve it.

Maps

- A map is a collection of pairs
  - A ***value****:* The object being stored
  - A ***key***: Another object used to look up and find the value.
- An array is a basic map implementation. You assign an index (a key) and a value
- The value can then be looked up by its key

Maps

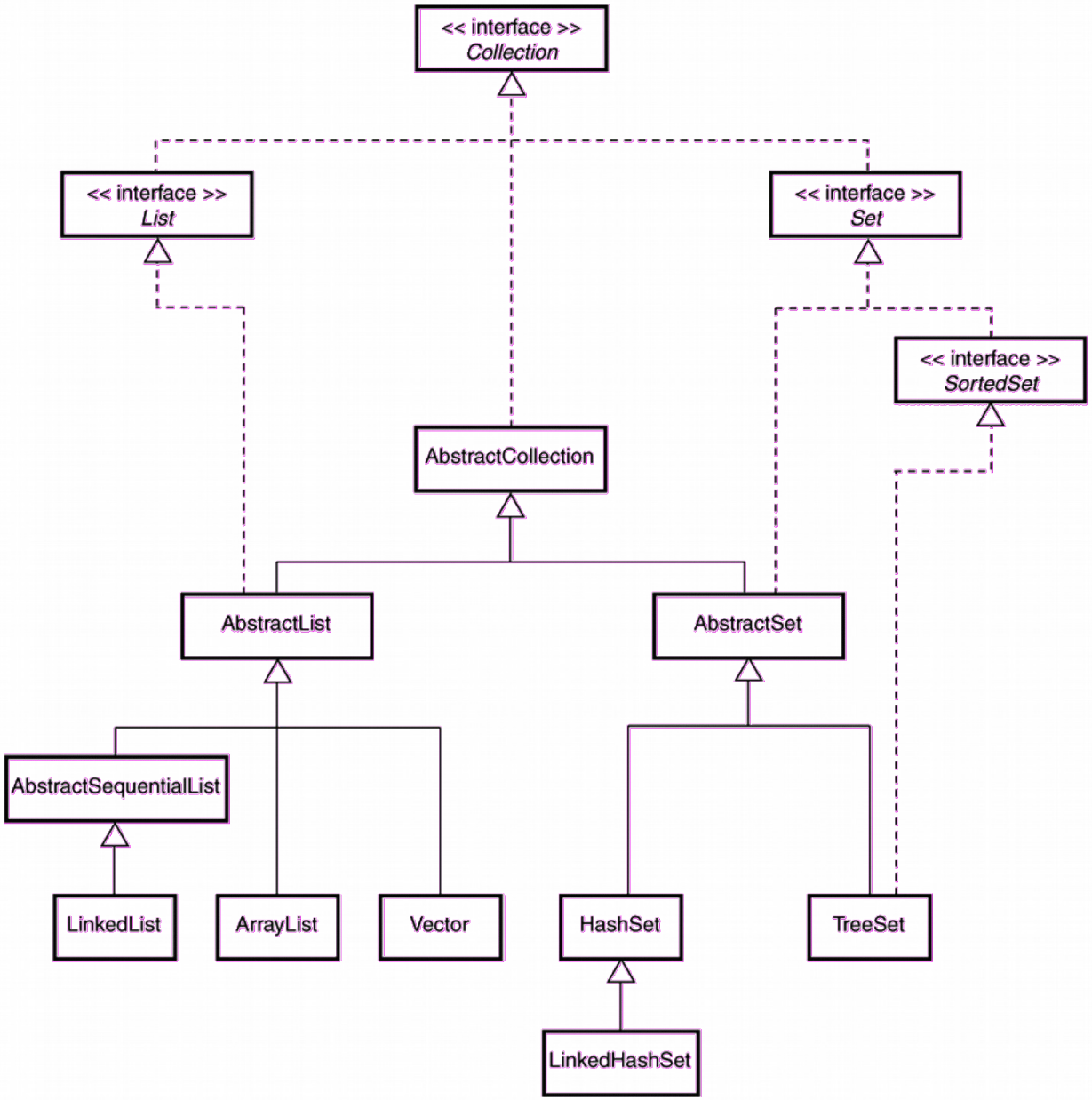
```
String[] stringArray = new String[5];

stringArray[3] = "A";
stringArray[4] = "B";

System.out.println(stringArray[3]);
System.out.println(stringArray[4]);
```

A value can be stored under a “key”. In an array this is an integer

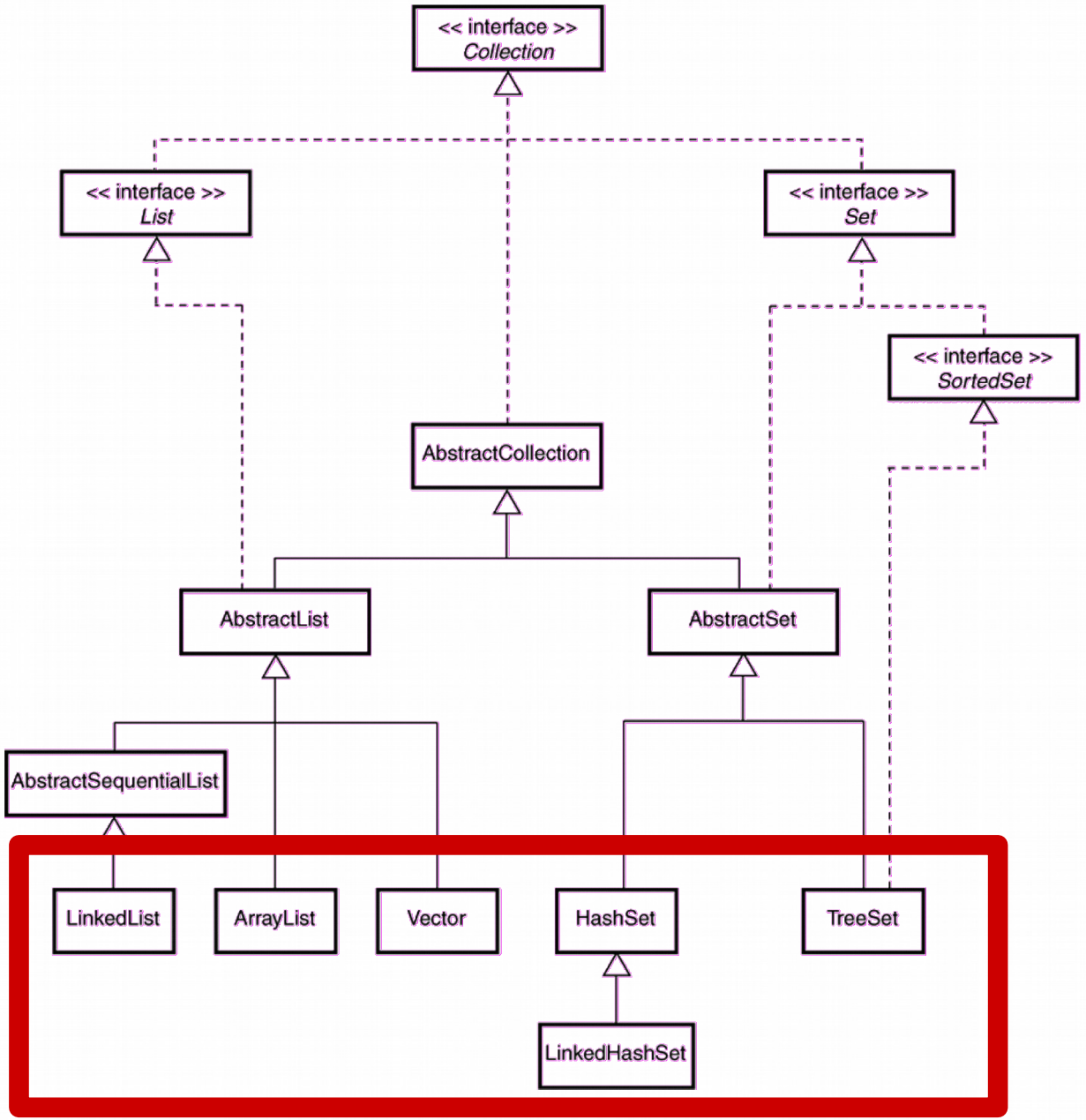
The value can be retrieved at a later date by the same key

- However, arrays only allow integers as keys, maps allow any object (E.g. strings are particularly useful!)
- Maps are useful for modelling real world relationships when information needs to be looked up.
- A person could be stored with their name as a key

Java Collections



Java Collections



Java Collections

- Lists and sets are fundamentally very similar. They are a set of values without a key
- Lists have an *index* but this is never determined by the programmer
- Maps are fundamentally different from Lists and Sets so use a different interface.

Collection interface

Method	Description
add(o : E) : boolean	Adds an object o to the Collection. The method returns true if o is successfully added to the collection, false otherwise.
clear() : void	Removes all elements from the collection.
contains(o : Object): boolean	Returns true if o is an element of the collection, false otherwise.
isEmpty() : boolean	Returns true if there are no elements in the collection, false otherwise.
iterator() : Iterator<E>	Returns an object called an iterator that can be used to examine all elements stored in the collection.
remove(o : Object) : boolean	Removes the object o from the collection and returns true if the operation is successful, false otherwise.
size() : int	Returns the number of elements currently stored in the collection.

Collection interface

Method	Description
add(o : E) : boolean	Adds an object o to the Collection. The method returns true if o is successfully added to the collection, false otherwise.
clear() : void	Removes all elements from the collection.
contains(o : Object): boolean	Returns true if o is an element of the collection, false otherwise.
isEmpty() : boolean	Returns true if there are no elements in the collection, false otherwise.
iterator() : Iterator<E>	Returns an object called an iterator that can be used to examine all elements stored in the collection.
remove(o : Object) : boolean	Removes the object o from the collection and returns true if the operation is successful, false otherwise.
size() : int	Returns the number of elements currently stored in the collection.

Lists

- The list interface provides methods that are relevant to all Lists. It extends the *collection* interface
- Things like:
  - Adding to the the list
  - Retrieving an element from the list
  - Removing and item from the list

Types of List

- There are 3 types of list:
  - ArrayList
  - Vector
  - LinkedList
- These all have the same *interface* but work differently internally.
- The difference is their performance characteristics. They have uses at different times

List interface methods

add(index:int, el:E) : void	Adds the element el to the collection at the given index. Throws IndexOutOfBoundsException if index is negative, or greater than the size of the list.
get(index:int):E	Returns the element at the given index, or throws IndexOutBoundsException if index is negative or greater than or equal to the size of the list.
indexOf(o:Object):int	Returns the least (first) index at which the object o is found; returns -1 if o is not in the list.
lastIndexOf(o:Object):int	Returns the greatest (last) index at which the object o is found; returns -1 if o is not in the list.
listIterator():ListIterator<E>	Returns an iterator specialized to work with List collections.
remove(index:int):E	Removes and returns the element at the given index; throws IndexOutOfBoundsException if index is negative, or greater than or equal to the size of the list.
set(index:int, el:E):E	Replaces the element at index with the new element el.

ArrayList and Vector

- **ArrayList** and **Vector** are array-based lists. Internally they use arrays to store their elements
- Whenever the array gets full a new, bigger array is created and the elements are copied to the new array
- **Vector** has higher overhead than **ArrayList** to make it safe to use in programs with multiple threads
- (In anything for this module, **ArrayList** is preferrable!)

LinkedList

- Array-based lists have high overhead when elements are being inserted because the array has to be resized and the elements have to be moved around.
- E.g. inserting an element at position 12 means anything after 12 in the list must be moved up one
- This operation can be slow on large lists

LinkedList

- However, looking up a specific index in a LinkedList is slower!
- list.get(123) is fast on an arraylist because it just looks up the index 123
- In a LinkedList the LinkedList must iterate over its elements until it reaches position 123 which is significantly slower!

LinkedList

- A LinkedList stores elements in a way that eliminates this overhead by storing elements along with their successor/predecessor
- When an element is added in the middle of the list only 3 changes need to happen:
  - The element before the new one has its successor updated
  - The element after the new one has its predecessor updated
  - The element being inserted has its successor and predecessor set
- This is a lot faster than an ArrayList as elements don't need moving around!

LinkedList vs ArrayList

- LinkedLists are better if you have a lot of write operations and only retrieve elements by iterating over it
- ArrayLists are better if you need to read elements from specific positions
- ****Most of the time you will notice no difference!**

  An ArrayList will almost always be fine!**

Polymorphism

- Because lists all use the same *interface*, you can use polymorphism and change your mind at any time! You can change a LinkedList to an ArrayList at any time by changing one line where its defined

Lists

```
import java.util.*;
public class Test {
   public  static void main(String [ ] args)  {
       List<String> nameList = new ArrayList<String> ();
       String[] names = {"Ann", "Bob", "Carol"};

       // Add to arrayList
       for (int k = 0; k < names.length; k++)
           nameList.add(names[k]);

       // Display name list
       for (int k = 0; k < nameList.size(); k++)
           System.out.println(nameList.get(k));
    }
}
```

Define the list

Add elements the list

Retrieve elements

```
Output:
Ann
Bob
Carol
```

Lists

```
import java.util.*;
public class Test {
   public  static void main(String [ ] args)  {
       List<String> nameList = new LinkedList<String> ();
       String[] names = {"Ann", "Bob", "Carol"};

       // Add to arrayList
       for (int k = 0; k < names.length; k++)
           nameList.add(names[k]);

       // Display name list
       for (int k = 0; k < nameList.size(); k++)
           System.out.println(nameList.get(k));
    }
}
```

Because ArrayList and LinkedList share the same interface it doesn't matter if you change your mind and update the type at any time

Lists

```
public class Test {
   public  static void main(String [ ] args)  {
       List<String> nameList = new ArrayList<String> ();
       String [ ] names = {"Ann", "Bob", "Carol"};

       // Add to arrayList
       for (int k = 0; k < names.length; k++)
           nameList.add(names[k]);

       // Display name list
       for (int k = 0; k < nameList.size(); k++)
           System.out.println(nameList.get(k));


       method(nameList);
    }

   public static void method(List<String> list) {
       list.add("Dave");
   }
}
```

As lists are *interchangable* you should always type hint the interface name rather than the full type name

Lists

```
public class Test {
   public  static void main(String [ ] args)  {
       List<String> nameList = new ArrayList<String> ();
       String [ ] names = {"Ann", "Bob", "Carol"};

       // Add to arrayList
       for (int k = 0; k < names.length; k++)
           nameList.add(names[k]);

       // Display name list
       for (int k = 0; k < nameList.size(); k++)
           System.out.println(nameList.get(k));


       method(nameList);
    }

   public static void method(LinkedList<String> list) {
       list.add("Dave");
   }
}
```

Otherwise you cannot change your mind!

This will break because the Method specifically requires A LinkedList and namesList Is an ArrayList

Iterators

- An iterator is an object that is associated with a collection.
- It provides methods for fetching all of the elements in the collection one at a time in order

- Iterators have a method for removing the last item fetched from the collection

Iterators

- An iterator is an object that is returned from a collection's iterator() method
- An iterator has a *cursor position* although this is internal and hidden from you.
- An iterator has the following methods:
  - hasNext() returns true if there are elements remaining the collection
  - next() returns the next available element and increments the cursor
  - remove() remove the current element

Iterator remove()

- The **remove()** method removes the latest element returned by the call to **next()**
- Because of this, the **remove()** method can only be called once per call to **next()**

Using an Iterator

```
public  static void main(String [ ] args)  {
       String[] names = {"Ann", "Bob", "Carol"};

       List<String> nameList = new ArrayList<String> ();

       for (int i = 0; i < names.length; i++) {
          nameList.add(names[i]);

       }

       Iterator<String> it = nameList.iterator();
       while (it.hasNext()) {
          System.out.println(it.next());
       }
}
```

```
Output:
Ann
Bob
Carol
```

Create the List and add some elements to it

Get the iterator

Loop while there is another element

Get the current element

Iterators

- Each collection provides an Iterator
- This means you can have a standard method of iterating over any collection and don't need to rely on the .get method on the list
- Iterators overcome the performance issues of LinkedLists when using them in a loop

Iterators

```
public  static void main(String [ ] args)  {
       String[] names = {"Ann", "Bob", "Carol"};

       List<String> nameList = new LinkedList<String> ();

       for (int i = 0; i < names.length; i++) {
          nameList.add(names[i]);

       }

       Iterator<String> it = nameList.iterator();
       while (it.hasNext()) {
          System.out.println(it.next());
       }

       for (int i = 0; i < nameList.size(); i++) {
           System.out.println(nameList.get(i));
       }

}
```

This is slow. Each time **get(i)** is called the list Is iterated over internally!

Iterators

- Be careful when using iterators (or while loops generally) as it's easy to accidentally cause an infinite loop. If **next()** is never called, the cursor will never get moved and **hasNext()** will always return true!

```
Iterator<String> it = nameList.iterator();
while (it.hasNext()) {
   System.out.println(it.next());
}

Iterator<String> it = nameList.iterator();
while (it.hasNext()) {

}
```

Iterators

- Because of these performance considerations it's good practice to always use the iterator when iterating over a list (or any other collection)
- This is because each Iterator is specific to the type being iterated over
- Some iterators provide extra methods

Iterators

- The ListIterator provides methods for hasPrevious() and hasNext();
- This is useful for iterating over the array in reverse order. Note that this is not available for other collections!

```
Iterator< String[] names = {"Ann", "Bob", "Carol"};

List<String> nameList = new LinkedList<String> ();

for (int i = 0; i < names.length; i++) {
   nameList.add(names[i]);
}

 Iterator<String> it = nameList.iterator();
while (it.hasPrevious()) {
   System.out.println(it.previous());
}
```

```
Output:
Carol
Bob
Ann
```

Iterators

- The enhanced for loop can be used with any collection
- It automatically recognises the *iterator* and uses it internally

```
 List<String> nameList = new LinkedList<String> ();
nameList.add("Ann");
nameList.add("Bob");
nameList.add("Carol");

for (String name : nameList) {
System.out.println(name);
}
```

Sets

- **Sets** are collections that store elements but have no notion of the position of an element within the collection

- Sets do not allow duplicate elements
- Sets generally are not as useful as other collections

Sets

- Java provides the following Set classes:
  - TreeSet
  - HashSet
  - LinkedHashSet

TreeSet

- A TreeSet orders the elements that are stored but is slower than a HashSet and LinkedHashSet

```
TreeSet<Integer> tree = new TreeSet<Integer>();
tree.add(12);
tree.add(63);
tree.add(34);
tree.add(12);
tree.add(45);

Iterator<Integer> iterator = tree.iterator();
while (iterator.hasNext()) {
    System.out.println(iterator.next());
}
```

```
Output:
12
34
45
63
```

The elements have been sorted and 12 only appears once

HashSet

- HashSets make no guarantee of order but are faster than tree sets. They should be used when order is not important

```
Set<Integer> tree = new HashSet<Integer>();
tree.add(12);
tree.add(63);
tree.add(34);
tree.add(12);
tree.add(45);

Iterator<Integer> iterator = tree.iterator();
while (iterator.hasNext()) {
    System.out.println(iterator.next());
}
```

```
Output:
34
12
45
63
```

Order is seemingly random and 12 only appears once

HashSet

- LinkedHashSets are faster than TreeSets but slower than HashSets but preserve the order elements were added in

```
Set<Integer> tree = new LinkedHashSet<Integer>();
tree.add(12);
tree.add(63);
tree.add(34);
tree.add(12);
tree.add(45);

Iterator<Integer> iterator = tree.iterator();
while (iterator.hasNext()) {
    System.out.println(iterator.next());
}
```

```
Output:
34
12
45
63
```

Order is the same as elements were added in

Maps

- Maps store data using Key->Value mappings
- Keys and values can be any types
- Strings are useful as keys as they are readable in the code
- Map is a generic interface Map<K,V>
- Note that Map takes **two** generic types. K for the key and V for the value part of the mapping.
- These can be the same, but you must provide both types

Maps

- There are 3 types of Maps that store their keys as Sets. As such, they mirror the Set types:
  - **HashMap** which makes no guarantee of sort order
  - **LinkedHashMap** which stores element according to the order they were inserted
  - **TreeMap** which stores element according to the natural order of keys, or the order specified by the programmer

Maps

- Maps store elements under a key which can then be retrieved using the same key
- You can specify any types for either Key or Value
- This is useful for real world relationships between objects
- E.g. a person's name and their age
- The functionality is very similar to an array but you don't have to use an integer as the key and the do not have a fixed size

Maps

```
Map<String,Integer> people = new HashMap<String,Integer>();
people.put("Ann", 23);
people.put("Bob", 29 );
people.put("Carol", 35);
people.put("Dave", 20);


System.out.println(people.get("Dave"));
```

```
Output:
20
```

Two generic types
one for keys
one for values

Note the use of **put()**
Instead of add!

Store a value
under a key

Which can then be
Retrieved by the key

HashMap

```
Map<String,Integer> map = new HashMap<String,Integer>();
map.put("C", 12);
map.put("E", 63);
map.put("A", 72);
map.put("B", 45);
map.put("T", 22);


Iterator<String> iterator = map.keySet().iterator();
while (iterator.hasNext()) {
    String key = iterator.next();
    System.out.println("key: " + key +
            " value: " + map.get(key));
}
```

Note the use of **put()** Instead of add!

The iterator is on the set of keys

```
Output:
key: A value: 72
key: B value: 45
key: C value: 12
key: T value: 22
key: E value: 63
```

The order isn't preserved

TreeMap

```
Map<String,Integer> map = new LinkedHashMap<String,Integer>();
map.put("C", 12);
map.put("E", 63);
map.put("A", 72);
map.put("B", 45);
map.put("T", 22);


Iterator<String> iterator = map.keySet().iterator();
while (iterator.hasNext()) {
    String key = iterator.next();
    System.out.println("key: " + key +
            " value: " + map.get(key));
}
```

Note the use of **put()** Instead of add!

The iterator is on the set of keys

```
Output:
key: C value: 12
key: E value: 63
key: A value: 72
key: B value: 45
key: T value: 22
```

The order is the “natural” order of the **keys** with strings, Alphabetical order

TreeMap

```
Map<String,Integer> map = new TreeMap<String,Integer>();
map.put("C", 12);
map.put("E", 63);
map.put("A", 72);
map.put("B", 45);
map.put("T", 22);


Iterator<String> iterator = map.keySet().iterator();
while (iterator.hasNext()) {
    String key = iterator.next();
    System.out.println("key: " + key +
            " value: " + map.get(key));
}
```

Note the use of **put()** Instead of add!

The iterator is on the set of keys

```
Output:
key: A value: 72
key: B value: 45
key: C value: 12
key: E value: 63
key: T value: 22
```

The order is the same as the order the elements were added in

Comparators

- TreeSets and TreeHashMaps sort elements in "natural" order by default
- However, you can provide a sorting implementation
- This is known as a ***Comparator***
- A Comparator is a class which implements the Comparator interface
- It has one method compare() which takes two elements from the collection and returns a -1 or +1 value depending on whether one should be before or after the other

Comparator Example - Car

- Consider a Car Class which stored a Car's Registration and Mileage

```
public class Car {
    private String registration;
    private int mileage;

    public Car(String registration, int mileage) {
        this.registration = registration;
        this.mileage = mileage;
    }

    public int getMileage() {
        return this.mileage;
    }

    public String getRegistration() {
        return this.registration;
    }
}
```

Car Class

- You can store a collection of cars using a TreeSet and iterate over them:

```
TreeSet<Car> tree = new TreeSet<Car>();
tree.add(new Car("ABC12", 12000));
tree.add(new Car("BN133", 14000));
tree.add(new Car("CA173", 1500));
tree.add(new Car("ZN993", 20000));

Iterator<Car> iterator = tree.iterator();
while (iterator.hasNext()) {
    Car current = iterator.next();
    System.out.println(current.getRegistration()
                + " has " +  current.getMileage() + " miles");

}
```

Car Class

- However, there is no "natural" order for Cars because they are a complex object
- It's possible to define a Comparator class to sort the cars by their mileage

Car Class

- To enable the TreeSet to sort the Cars you must write a Comparator. This will sort the cars by Mileage

```
import java.util.Comparator;

public class CarComparator implements Comparator<Car> {
    public int compare(Car car1, Car car2) {
        if (car1.getMileage() < car2.getMileage()) {
            return -1;
        }
        else {
            return 1;
        }
    }
}
```

Type being compared in implements declaration

Some logic to return -1 Or 1 depending on whether Car1 should be before (-1) Or after (+1) Car2

Car Class

- To enable the TreeSet to sort the Cars you must write a Comparator. This will sort the cars by Mileage

```
TreeSet<Car> tree = new TreeSet<Car>(new CarComparator());
tree.add(new Car("ABC12", 12000));
tree.add(new Car("BN133", 14000));
tree.add(new Car("CA173", 1500));
tree.add(new Car("ZN993", 20000));

Iterator<Car> iterator = tree.iterator();
while (iterator.hasNext()) {
    Car current = iterator.next();
    System.out.println(current.getRegistration()
                + " has " +  current.getMileage()
 + " miles");

}
```

Pass an instance of the Comparator to the Set's Constructor

The iterator will now use the comparator to sort the elements

Car Class

- To enable the TreeSet to sort the Cars you must write a Comparator. This will sort the cars by Mileage

```
TreeSet<Car> tree = new TreeSet<Car>(new CarComparator());
tree.add(new Car("ABC12", 12000));
tree.add(new Car("BN133", 14000));
tree.add(new Car("CA173", 1500));
tree.add(new Car("ZN993", 20000));

Iterator<Car> iterator = tree.iterator();
while (iterator.hasNext()) {
    Car current = iterator.next();
    System.out.println(current.getRegistration()
                + " has " +  current.getMileage()
 + " miles");

}
```

```
Output:
CA173 has 1500 miles
ABC12 has 12000 miles
BN133 has 14000 miles
ZN993 has 20000 miles
```