

## Chapter 6 Collections

### Enum

- ▶ An enum type is a special data type that enables for a variable to be a set of predefined constants
  - Enum types are effectively final
  - no accessible constructor
  - clients can neither create instances or extend it
- ▶ Enum types are instance controlled
- ▶ Enums are generalization of Singletons
- ▶ Singleton is a single element enum

## Enum

- ▶ An enum type is a special data type that enables for a variable to be a set of predefined constants
  - Enum types are effectively final
  - no accessible constructor
  - clients can neither create instances or extend it

```
enum Color{
    RED, GREEN, BLUE
}

Color color = Color.RED;
```

3

## use class to control objects

```
▶ class Color{
    public static final Color RED = new Color("red");
    public static final Color GREEN = new Color("green");
    public static final Color BLUE = new Color("blue");
    private String name;
    private Color(String name){
        this.setName(name);
    }
    public String getName() {
        return this.name;
    }
}

▶ public class ColorDemo {
    public static void main(String args[]){
        Color c1 = Color.RED;
        System.out.println(c1.getName());
    }
}
```

**Output:**  
red

4

## use enum

```

▶ public enum Color{
▶     RED, GREEN, BLUE ;
▶ }

▶ public class GetEnumContent {
▶     public static void main(String args[]){
▶         Color c = Color.BLUE ;
▶         System.out.println(c);
▶     }
▶ }

```

**Output:**  
**BLUE**

5

## Methods in enum

- ▶ All enum types are sub types of java.lang.Enum
- ▶ Methods of java.lang.Enum

<i>values()</i>	Obtain all the constants of an enum type
<i>name()</i>	Returns the name of this enum constant
<i>ordinal()</i>	Returns the ordinal of this enumeration constant

String	<i>name()</i>	Returns the name of this enum constant, exactly as defined in the source code.
int	<i>ordinal()</i>	Returns the ordinal of this enumeration constant (its position in the enum declaration; the initial constant is assigned an ordinal of zero).
String	<i>toString()</i>	Returns the name of this enum constant, as contained in the source code.
static <T extends Enum<T>> T valueOf(Class<T> enumType, String name)		Returns the enum constant of the specified enum type with the specified name.

6

## Enum syntax : foreach

```

public class PrintDemo {
    public static void main(String args[]){
        for(Color c : Color.values()){
            System.out.println(c);
        }
    }
}

```

**Output:**  
RED  
GREEN  
BLUE

7

## Enum syntax : switch

```

public class SwitchPrintDemo {
    public static void main(String args[]){
        for(Color c : Color.values()){
            print(c);
        }
    }
    public static void print(Color color){
        switch(color){
            case RED:{
                System.out.println("red");
                break ;
            }
            case GREEN:{
                System.out.println("green");
                break ;
            }
            case BLUE:{
                System.out.println("blue");
                break ;
            }
            default :{
                System.out.println("undefined");
                break ;
            }
        }
    }
}

```

**Output:**  
red  
green  
blue

8

## Enum Example

```

enum Color{
    RED, GREEN, BLUE ;
}

public class GetEnumInfo {
    public static void main(String args[]){
        for(Color c : Color.values() ){
            System.out.println(c.ordinal() + " --> " + c.name());
        }
    }
}

```

**Output:**  
 0 --> RED  
 1 --> GREEN  
 2 --> BLUE

9

## Java generic

- Java Generic methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods, or with a single class declaration, a set of related types, respectively.



10

## Cast Exceptions at Runtime

```
public class OldBox {
    Object data;
    public OldBox(Object data) {
        this.data = data;
    }
    public Object getData() {
        return data;
    }

    public static void main(String[] args){
        OldBox intBox = new OldBox(42);
        int x = (Integer) intBox.getData();
        OldBox strBox = new OldBox("Hi");
        String s = (String) strBox.getData();
        int y = (Integer) strBox.getData();
        intBox = strBox;
    }
}
```

Problems @ Javadoc Declaration Console Variables Breakpoints

<terminated> OldBox [Java Application] C:\Program Files\Java\jre1.8.0\_131\bin\javaw.exe (2018年4月26日 下午11:29:30)

Exception in thread "main" java.lang.ClassCastException: java.lang.String cannot be cast to java.lang.Integer  
at OldBox.main(OldBox.java:15)

## Naïve Solution

```
public class IntBox {
    Integer data;
    public IntBox(Integer data) {
        this.data = data;
    }
    public Integer getData() {
        return data;
    }
}

public class StrBox {
    String data;
    public StrBox(String data) {
        this.data = data;
    }
    public String getData() {
        return data;
    }
}
```

Usage limited  
Infinite many classes possible

## Passing Parameters to Methods

```
public class Sum {  
    public static int sum_0_1() {  
        return (0 + 1);  
    }  
  
    public static int sum_15_22() {  
        return (15 + 22);  
    }  
}
```

Infinite many methods

```
public class NewSum {  
    public static int sum(int m, int n) {  
        return m + n;  
    }  
}
```

## Java Generic: Key Idea

- ▶ Parameterize type definitions
  - Parameterized classes and methods
- ▶ Provide type safety
  - Compiler performs type checking
  - Prevent runtime cast errors

## Parameterized Classes

```
public class Box<E> {
    E data;

    public Box(E data) {
        this.data = data;
    }

    public E getData() {
        return data;
    }
}
```

- E refers to a particular type
- The constructor takes an object of type E, not any object
- To use this class, E must be replaced with a specific class

15

## Parameterized Classes

```
Box<Integer> intBox = new Box<Integer>(42);
int x = intBox.getData(); // no cast needed
```

```
Box<String> strBox = new Box<String>("Hi");
String s = strBox.getData(); // no cast needed
```

```
String s = (String) intBox.getData();
int y = (Integer) strBox.getData();
intBox = strBox;
```

```
15 Box<Integer> intBox = new Box<Integer>(42);
16 int x = intBox.getData(); // no cast needed
17
18 Box<String> strBox = new Box<String>("Hi");
19 String s = strBox.getData(); // no cast needed
20
21 String s = (String) intBox.getData();
22 int y = (Integer) strBox.getData();
23 intBox = strBox;
```

Runtime errors now converted to compile time errors

16



## Parameterized Classes: Syntax Note

**A class can have multiple parameters:**

```
public class Stuff<A,B,C> { ... }
```

**Subclassing parameterized classes allowed**

```
/* Extending a particular type */
class IntBox extends Box<Integer> { ... }
```

**Or**

```
/* Extending a parameterized type */
class SpecialBox<E> extends Box<E> { ... }
```

**SpecialBox<String> is a subclass of Box<String>.**

```
/* Following assignment is legal */
```

```
Box<String> sbox = new SpecialBox<String>("Hi");
```

17

## Parameterized Classes in Methods

A parameterized class is a type just like any other class. It can be used in method input types and return types.

```
public class Box<E> {
    E data;
    public Box(E data) {
        this.data = data;
    }
    public E getData() {
        return data;
    }
    public void copyFrom(Box<E> b) {
        this.data = b.getData();
    }
}
//We have added an infinite number of types of Boxes
//by writing a single class definition
```

18

## Generic Methods

- Generic methods allow type parameters to be used to express dependencies among the types of one or more arguments to a method and/or its return type.

```
class ArrayGen{
    public static <T> T getMiddle(T[] t){
        return t[t.length/2];
    }
}

public class GenericMethods {
    public static void main(String[] args) {
        String s[] = { "we", "are", "studying", "Java", "language!" };
        String middle1 = ArrayGen.<String>getMiddle(s);
        System.out.println(middle1);
        Integer i[] = { new Integer(1), 2, 3, 4 };
        Integer middle2 = ArrayGen.getMiddle(i);
        System.out.println(middle2);
    }
}
```

19

## Generic Methods

```
public class GenericMethods {
    public <T> T aMethod(T x) {
        return x;
    }

    public static void main(String[] args) {
        GenericMethods gm = new GenericMethods();
        int k = gm.aMethod(5);
        String s = gm.aMethod("abc");
    }
}

public class Bar<T> {
    public T aMethod(T x) {
        return x;
    }

    public static void main(String[] args) {
        Bar<Integer> bar = new Bar<Integer>();
        int k = bar.aMethod(5);
        String s = bar.aMethod("abc");
    }
}
```

## Generic Interfaces

```
interface GenInterfaceBox<T>{
    boolean contains(T t);
}

class GenClassBox<T> implements GenInterfaceBox<T> {
    T[] elements;

    GenClassBox(T[] elements) {
        this.elements = elements;
    }

    public boolean contains(T t) {
        for (T x : elements)
            if (x.equals(t))
                return true;
        return false;
    }
}
```

21

## Bounded Parameterized Types

Sometimes we want restricted parameterization of classes.  
 We want a box, called `MathBox` that holds only `Number` objects.  
 We can't use `Box<E>` because `E` could be anything.  
 We want `E` to be a subclass of `Number`.

```
public class MathBox<E extends Number> extends Box<Number> {
    public MathBox(E data) {
        super(data);
    }
    public double sqrt() {
        return Math.sqrt(getData().doubleValue());
    }
}
```

22

## Bounded Parameterized Types

```
public class MathBox<E extends Number> extends Box<Number> {
    public MathBox(E data) {
        super(data);
    }
    public double sqrt() {
        return Math.sqrt(getData().doubleValue());
    }
}
```

The `<E extends Number>` syntax means that the type parameter of `MathBox` must be a subclass of the `Number` class. We say that the type parameter is bounded.

```
new MathBox<Integer>(5); //Legal
new MathBox<Double>(32.1); //Legal
new MathBox<String>("No good!"); //Illegal
```

23

## Bounded Parameterized Types

Inside a parameterized class, the type parameter serves as a valid type.

So the following is valid.

```
public class OuterClass<T> {
    private class InnerClass<E extends T> {
        ...
    }
    ...
}
```

**Syntax note:** The `<A extends B>` syntax is valid even if `B` is an interface.

24

## Bounded Parameterized Types

Java allows multiple inheritance in the form of implementing multiple interfaces. So multiple bounds may be necessary to specify a type parameter. The following syntax is used then:

`<T extends A & B & C & ...>`

For instance:

```
interface A {
    ...
}
interface B {
    ...
}
class MultiBounds<T extends A & B> {
    ...
}
```

25

## Generics and Subtyping

We start to run into some new issues when we do some things that seem "normal". For instance, the following seems reasonable:

```
Box<Number> numBox = new Box<Integer>(31);
```

**Compiler comes back with an "Incompatible Type" error message.**

This is because `numBox` can hold only a `Number` object and nothing else, not even an object of type `Integer` which is a subclass of `Number`.

`Box<T>` is not a subclass of `Box<E>` even if `T` is a subclass of `E`.

```
//Consider the following lines of code
Box<String> strBox = new Box<String>("Hi");//1
Box<Object> objBox = strBox;//2 - compilation error
objBox.setData(new Object());//3
String s = strBox.getData();//4 - an Object to a String!
```

26

## Unbounded Wildcards

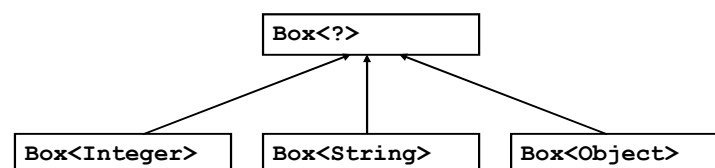
We want to write a method to print any `Box`.

```
public static void printBox(Box<Object> b) {  
    System.out.println(b.getData());  
}  
  
Box<String> strBox = new Box<String>("Hi");  
printBox(strBox); // compilation error  
  
public static void printBox(Box<?> b) {  
    System.out.println(b.getData());  
} // using unbounded wildcard
```

27

## Unbounded Wildcards

**`Box<?>` is a superclass of `Box<T>` for any `T`.**

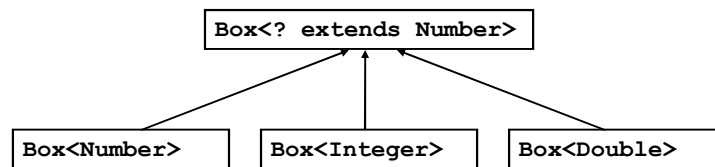


Unbounded wildcards are useful when writing code that is completely independent of the parameterized type.

28

## Upper Bounded Wildcards

A `Box` of any type which is a subtype of `Number`



```
Box<? extends Number> numBox = new Box<Integer>(31);
```

`<? extends E>` is called "*upper bounded wildcard*" because it defines a type that is bounded by the superclass `E`.

29

## Upper Bounded Wildcards

```
public class Box<E> {
    public void copyFrom(Box<E> b) {
        this.data = b.getData();
    }
}

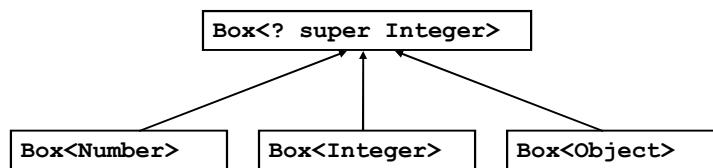
public class Box<E> {
    public void copyFrom(Box<? extends E> b) {
        this.data = b.getData(); //b.getData() is a
                                //subclass of this.data
    }
}

Box<Integer> intBox = new Box<Integer>(31);
Box<Number> numBox = new Box<Number>();
numBox.copyFrom(intBox);
```

30

## Lower Bounded Wildcards

A `Box` of any type which is a supertype of `Integer`



`<? super E>` is called a "*lower bounded wildcard*" because it defines a type that is bounded by the subclass `E`.

31

## Lower Bounded Wildcards

Suppose we want to write `copyTo()` that copies data in the opposite direction of `copyFrom()`.

`copyTo()` copies data from the host object to the given object.

This can be done as:

```
public void copyTo(Box<E> b) {
    b.data = this.getData();
}
```

Above code is fine as long as `b` and the host are boxes of exactly same type. But `b` could be a box of an object that is a superclass of `E`.

This can be expressed as:

```
public void copyTo(Box<? super E> b) {
    b.data = this.getData();
    //b.data() is a superclass of this.data()
}
```

```
Box<Integer> intBox = new Box<Integer>(31);
Box<Number> numBox = new Box<Number>();
intBox.copyTo(numBox);
```

32



## Generic example: type safety

```

> public class GenericTest {
>     public static void main(String[] args) {
>         Collection c = new ArrayList();
>         c.add("1");
>         c.add("2");
>         c.add(new Object());
>         Iterator i = c.iterator();
>         while (i.hasNext()) {
>             String s = (String) i.next();
>             System.out.println(s);
>         }
>     }
> }

```

Console Problems Javadoc Declaration Search

<terminated> GenericTest [Java Application] C:\Program Files (x86)\Java\jre7\bin\javaw.exe (2012-5-16 下午09:14:32)

h  
2  
Exception in thread "main" java.lang.ClassCastException: java.lang.Object cannot be cast to java.lang.String  
at GenericTest.main(GenericTest.java:13)

33

```

> import java.util.ArrayList;
> import java.util.Collection;
> import java.util.Iterator;

> public class GenericTest {
>     public static void main(String[] args) {
>         Collection<String> c = new ArrayList<String>();
>         c.add("1");
>         c.add("2");
>         c.add(new Object());
>         Iterator<String> i = c.iterator();
>         while (i.hasNext()) {
>             String s = i.next();
>             System.out.println(s);
>         }
>     }
> }

```

The method add(String) in the type Collection<String> is not applicable for the arguments (Object)

2 quick fixes available:

- Cast argument 'new Object()' to 'String'
- Change to 'addAll(...)'

Press 'F2' for focus

34

## Generic example: overloading

- ▶ Design method maximum to return maximum value from the three given values
- ▶ Public static `<T extends Comparable<T>> T`  
maximum(T x, T y, T z)

```
Public static Comparable maximum (Comparable
    x, Comparable y, Comparable z){
    Comparable max =x;
    .....
}
maximum(3, 4, 5);
maximum("3", "4", "5");
```

## Introduction

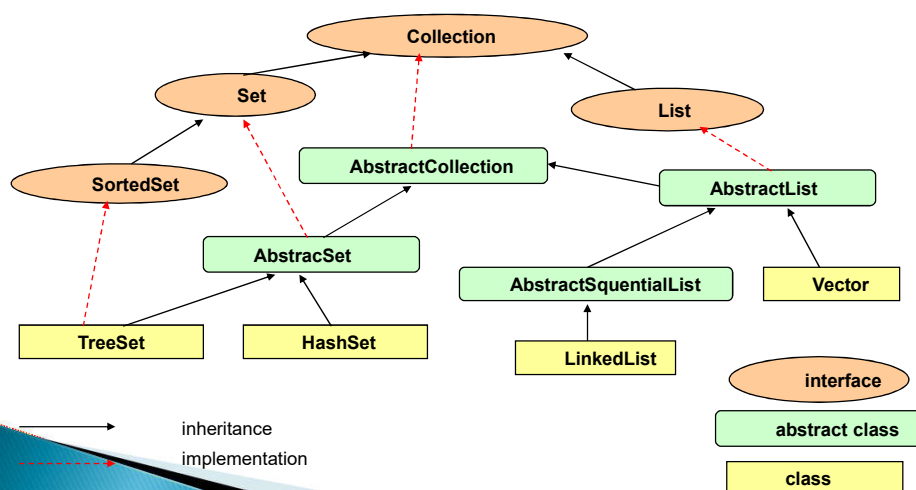
- ▶ A collection represents a group of objects, known as its elements, which may be stored in structured manner or not.
- ▶ The Java Collections Framework provides a well-designed set of interfaces and classes for storing and manipulating a collection such as addition, deletion, replacement, searching and traversal.
- ▶ It consists of three parts:
  - interfaces,
  - implementations
  - algorithms.

## Introduction

- ▶ There are four basic interfaces of the framework which forms a hierarchical
  - `Collection<E>`,
  - `Set<E>`,
  - `List<E>`,
  - `Map<K,V>`.

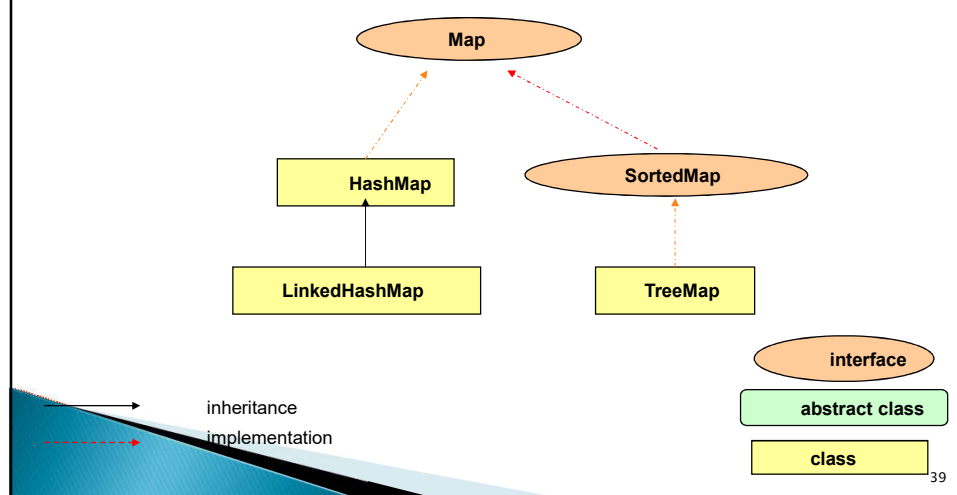
37

## Java collection



38

## Java collection



## Introduction

- ▶ The Collection<E> interface is the common ancestor.
- ▶ This parameterized interface represents a group of elements of some generic type E.
- ▶ E is a type parameter declared in the interface Collection definition.
- ▶ When you use this interface in your code, you may provide any type except primitive types as an argument.

## Introduction

- ▶ The Set<E> interface is a collection of type E that **cannot contain duplicate elements**.
- ▶ This interface models the mathematical set abstraction and is used to represent sets, such as the cards comprising a poker hand.
- ▶ List<E> is an ordered collection of type E and introduces **positional indexing**.
- ▶ The client of a List generally has precise control over it. In the list each element can be inserted and accessed by their integer index.

41

## Introduction

- ▶ The interfaces Map<K,V> manipulate **key-value pairs**.
- ▶ Each key can map to at most one value.
- ▶ A Map<K,V> object **cannot contain duplicate keys**.
- ▶ Here, K and V stand for any type except for the primitive types.

42

## Introduction

- ▶ The Java collection framework provides a set of standard classes that implement collection interfaces.
- ▶ Some of the classes provide full implementations that can be used directly and others are abstract classes, providing skeletal implementations that are used as starting points for creating your concrete collections.
  - HashSet extends AbstractSet for use with a hash table.
  - ArrayList implements a dynamic array by extending AbstractList.

43

## Introduction

Interface	Implementation	Historical
Set	HashSet, TreeSet	
List	ArrayList, LinkedList	Vector, Stack
Map	Hashtable, HashMap	Hashtable

44

## Introduction

- ▶ The collections framework defines several algorithms that perform useful computation, such as searching and sorting, on objects that implement collection interfaces.
- ▶ These algorithms are defined as static methods within the **Collections** class. The algorithms are reusable functionality in essence.

45

## Introduction

- ▶ The benefits provided by the collections framework include:
  - **High-performance**: The implementations for the fundamental collections are highly efficient.
  - **Easy-programming**: Useful data structures and algorithms are provided in the framework and can be employed easily. You don't have to write them yourself.
  - **Consistency**: Different types of collections work in a similar manner.
  - **Extendibility**: Extending and/or adapting a collection is easy. New data structures that conform to the standard collection interfaces are by nature reusable.

46

```

> public interface Collection<E> extends Iterable<E> {    // "E" stand for any type
> except for the primitive types.
> // Basic operations基础操作
> int size();
> boolean isEmpty();
> boolean contains(Object element);           //returns true if this collection
contains the specified element
> boolean add(E element);                     //optional, add an element into this collection
> boolean remove(Object element); //optional, Removes a single instance of the
specified element
> //from this collection
> Iterator<E> iterator();
>
> // Bulk operations批量操作
> boolean containsAll(Collection<?> c);       //"?" is a wildcard. It matches any
type.
> boolean addAll(Collection<? extends E> c);   //optional
> boolean removeAll(Collection<?> c);         //optional
> boolean retainAll(Collection<?> c);         //optional
> void clear();                               //optional
>
> // Array operations数组操作
> Object[] toArray();
> <T> T[] toArray(T[] a);
> }

```

47

## Set Interface

- ▶ Sets contain no pair of elements e1 and e2 such that **e1.equals(e2)**, and at most one null element.

48



## Set Interface

```

public interface Set<E> extends Collection<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element); // Returns true if this set contains the specified element
    boolean add(E o); // Adds the specified element to this set if it is not already present
    boolean remove(Object o); // Removes the specified element from this set if it is present

    Iterator<E> iterator();

    // Bulk operations
    boolean addAll(Collection<? extends E> c);
    boolean retainAll(Collection<?> c);
    boolean containsAll(Collection<?> c);
    boolean removeAll(Collection<?> c);
    void clear();
    // Array Operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}

```

49

## Set Interface

```

import java.util.Set;
import java.util.HashSet;
import java.util.Iterator;

public class SetTest {
    public static void main(String[] args) {
        Set<String> h = new HashSet<String>();
        System.out.println("Is the set empty? " + h.isEmpty());
        h.add("A");
        h.add("B");
        h.add("C");
        h.add("D");
        h.add("E");
        h.add(null);
        h.add("B");
        h.add(null);
        System.out.println("The number of elements in the set: " + h.size());
        System.out.println(h);
        System.out.println("Does the set contain null? " + h.contains(null));
    }
}

```

50

## Set Interface

```

> of the element. // The element removed is determined via the equals() method
> returned. If not found, false is returned.
> System.out.println("Is null removed? " + h.remove(null));
> System.out.println(h);
>
> // Traversal by an iterator
> Iterator<String> iter = h.iterator();
> while (iter.hasNext()) {
>     System.out.print(iter.next())
> }
> System.out.println();
> // Traversal by enhanced for loop
> for (String e : h) {
>     System.out.print(e);
> }
> System.out.println();
>
> // Removing all elements
> h.clear();
> System.out.println("Is the set cleared? " + h.isEmpty());
> }
}

```

The output is:  
 Is the set empty? true  
 The number of elements in the set  
 6  
 [null, D, E, A, B, C]  
 Does the set contain null? true  
 Is null removed? true  
 [D, E, A, B, C]  
 DEABC  
 DEABC  
 Is the set cleared? true

51

## Set Interface

### ► Bulk operations

- `A.addAll(B)` — The union of A and B.  $//C = A \cup B$
- `A.retainAll(B)` — The intersection of A and B.  $//C = A \cap B$
- `A.removeAll(B)` — The difference of A and B.  $//C = A - B$
- `A.containsAll(B)` — Is B subset of A?

52

## Set Interface

```

> import java.util.HashSet;
> import java.util.Set;
>
> public class SetDemo {
>     public static void main(String[] args) {
>         String[] a = { "A", "B", "C", "D", "E" };
>         String[] b = { "X", "Y", "E" };
>         Set<String> A = new HashSet<String>();
>         Set<String> B = new HashSet<String>();
>         for (String e : a) {
>             A.add(e);
>         }
>         for (String e : b) {
>             B.add(e);
>         }
>
>         Set<String> C;
>         C = new HashSet<String>(A);
>         // Union
>         C.addAll(B);
>         System.out.println(C);

```

53

## Set Interface

```

>         // Intersection
>         C = new HashSet<String>(A);
>         C.retainAll(B);
>         System.out.println(C);
>
>         // Difference
>         C = new HashSet<String>(A);
>         C.removeAll(B);
>         System.out.println(C);
>     }
> }

```

▶ It displays:

- ▶ [D, E, A, B, C, Y, X]
- ▶ [E]
- ▶ [D, A, B, C]

54

## Set Interface

- ▶ There are two general-purpose Set implementations:
  - HashSet
  - TreeSet
  - LinkedHashSet

55

## HashSet vs. TreeSet vs. LinkedHashSet

- ▶ HashSet is Implemented using a hash table. Elements are not ordered. The add, remove, and contains methods have constant time complexity  $O(1)$ .
- ▶ TreeSet is implemented using a tree structure (red-black tree in algorithm book). The elements in a set are sorted, but the add, remove, and contains methods have time complexity of  $O(\log(n))$ . It offers several methods to deal with the ordered set like first(), last(), headSet(), tailSet(), etc.
- ▶ LinkedHashSet is between HashSet and TreeSet. It is implemented as a hash table with a linked list running through it, so it provides the order of insertion. The time complexity of basic methods is  $O(1)$ .

56

## Example

```
class Car {
    int price;
    public Car(int price) {
        this.price = price;
    }
    public String toString() {
        return price + "Yuan";
    }
}

public class TestTreeSet {
    public static void main(String[] args) {
        TreeSet<Car> dset = new TreeSet<Car>();
        dset.add(new Car(200000));
        dset.add(new Car(100000));
        dset.add(new Car(300000));
        Iterator<Car> iterator = dset.iterator();
        while (iterator.hasNext()) {
            System.out.print(iterator.next() + " ");
        }
    }
}
```

Problems @ Javadoc Declaration Console (x) Variables Breakpoints

<terminated> TestTreeSet [Java Application] C:\Program Files\Java\jre1.8.0\_131\bin\javaw.exe (2018年5月3日 下午11:19:07)

Exception in thread "main" java.lang.ClassCastException: packone.Car cannot be cast to java.lang.Comparable

at java.util.TreeMap.compare(Unknown Source)

at java.util.TreeMap.put(Unknown Source)

at java.util.TreeSet.add(Unknown Source)

at packone.TestTreeSet.main(TestTreeSet.java:24)

57

## Example

- ▶ Because TreeSet is sorted, the Car object need to implement java.lang.Comparable's compareTo() method.

```
class Car implements Comparable<Car>{
    int price;
    public Car(int price) {
        this.price = price;
    }
    public String toString() {
        return price + "Yuan";
    }
    @Override
    public int compareTo(Car o) {
        return price - o.price;
    }
}
```

Problems @ Javadoc Declaration Console (x) Variables Breakpoints

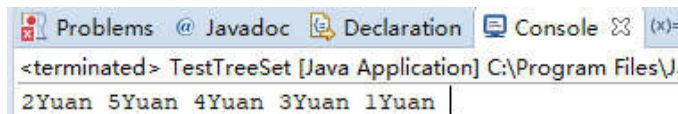
<terminated> TestTreeSet [Java Application] C:\Program Files\Java\jre1

100000Yuan 200000Yuan 300000Yuan

58

## Example

```
public static void main(String[] args) {
    LinkedHashSet<Car> dset = new LinkedHashSet<Car>();
    dset.add(new Car(2));
    dset.add(new Car(1));
    dset.add(new Car(3));
    dset.add(new Car(5));
    dset.add(new Car(4));
    Iterator<Car> iterator = dset.iterator();
    while (iterator.hasNext()) {
        System.out.print(iterator.next() + " ");
    }
}
```



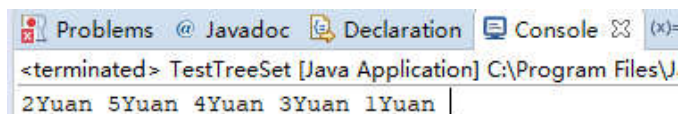
Problems @ Javadoc Declaration Console (x)=

<terminated> TestTreeSet [Java Application] C:\Program Files\J  
2Yuan 5Yuan 4Yuan 3Yuan 1Yuan |

59

## Example

```
public static void main(String[] args) {
    HashSet<Car> dset = new HashSet<Car>();
    dset.add(new Car(2));
    dset.add(new Car(1));
    dset.add(new Car(3));
    dset.add(new Car(5));
    dset.add(new Car(4));
    Iterator<Car> iterator = dset.iterator();
    while (iterator.hasNext()) {
        System.out.print(iterator.next() + " ");
    }
}
```



Problems @ Javadoc Declaration Console (x)=

<terminated> TestTreeSet [Java Application] C:\Program Files\J  
2Yuan 5Yuan 4Yuan 3Yuan 1Yuan |

60

## List Interface

- ▶ **Positional access** — manipulates elements based on their numerical position in the list.
- ▶ **Search** — searches for a specified object in the list and returns its numerical position.
- ▶ **Iteration** — The Iterator associated with a List returns its members in the order that they were entered taking into account additions/removals/replacements.
- ▶ **Range-view** — performs arbitrary range operations on the list.

61

## List Interface

- ▶ `public interface List<E> extends Collection<E> {`
- ▶ `// Basic operations`
- ▶ `int size();`
- ▶ `boolean isEmpty();`
- ▶ `boolean contains(Object element);` `//Returns true if this set contains the specified element`
- ▶ `boolean add(E element)` `//Appends the specified element to the end of this list`
- ▶ `void add(int index, E element)` `//Inserts the specified element at the specified position`
- ▶ `boolean remove(Object o)` `//Removes the first occurrence in this list of the specified element`
- ▶ `E remove(int index)` `//Removes the element at the specified position in this list`
- ▶ `E get(int index);` `//Returns the element at the specified position in this list`
- ▶ `E set(int index, E element)` `//Replaces the element at the specified position in this list with the`
- ▶ `//specified element`

62

## List Interface

```

> // Search
> int indexOf(Object o);
> int lastIndexOf(Object o);
> boolean contains(Object o)           //Returns true if this list contains the specified
    element
>
> // Iteration
> ListIterator<E> listIterator();       //Supplies not only the standard hasNext() and
    next() methods associated
> //with an Iterator, but also hasPrevious() and previous() methods
> // if you want to go back and forth at will.
> ListIterator<E> listIterator(int index);
>
> // Range-view
> List<E> subList(int from, int to);
>
> //Transformation
> Object[] toArray()                   //Convert a List into an array. You can also convert an array
    to a List using
> // the Arrays.asList(Object[]) method.
> }

```

63

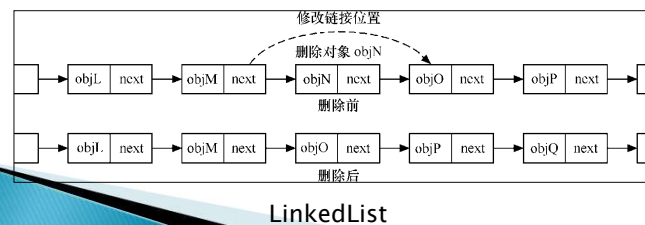
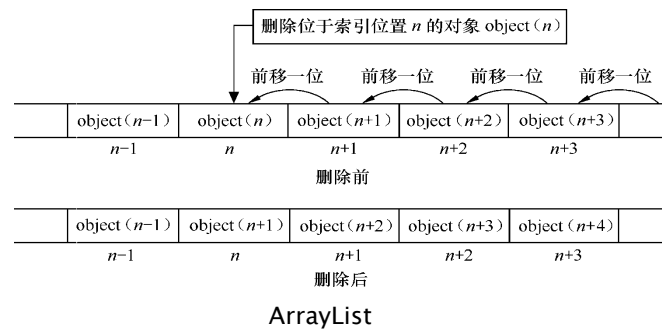
## List Interface

- ▶ The Java collection framework contains two general-purpose List implementations:
  - ArrayList
  - LinkedList.
- ▶ Also, Vector has been adapted historically to implement List.

64



## ArrayList and LinkedList



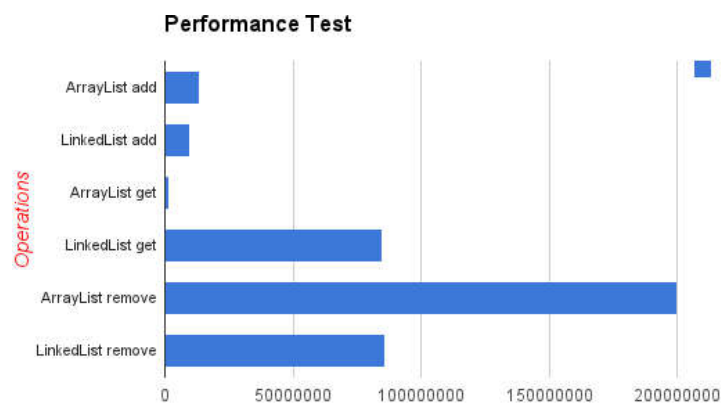
65

## List Interface

- ▶ **ArrayList** is implemented as a resizable array. As more elements are added to ArrayList, its size is increased dynamically. Its elements can be accessed directly by using the get and set methods, since ArrayList is essentially an array.
- ▶ **LinkedList** is implemented as a double linked list. Its performance on add and remove is better than ArrayList, but worse on get and set methods.
- ▶ **Vector** is similar with ArrayList, but it is synchronized.

66

## List Interface



67

## List Interface

```

1. import java.util.ArrayList;
2. import java.util.Arrays;
3. import java.util.List;
4. import java.util.ListIterator;
5.
6. public class ArrayListDemo {
7.     public static void main(String[] args) {
8.         String[] items = {"A", "B", "C", "D", "E", null, "F"};
9.         List<String> a = new ArrayList<String>(Arrays.asList(items));
10.        System.out.println(a);
11.        System.out.println("The element at 5: " + a.get(5));
12.        System.out.println("Replace the element at 5 with 'X': "
13.            + a.set(5, "X"));
14.        System.out.println(a);
15.        //Adding elements in the middle of a List
16.        System.out.println("Insert 'Y' before 6!");
17.        a.add(6, "Y");
18.        System.out.println(a);

```

68

## List Interface

```

19.      System.out.println("Remove the element at 3: " + a.remove(3));
20.      System.out.println(a);
21.
22.      // Traveling forward
23.      ListIterator<String> iter = a.listIterator();
24.      while (iter.hasNext()) {
25.          System.out.print(iter.next());
26.      }
27.      System.out.println();
28.
29.      //traveling backward
30.      //a.size() indicates the first element to be returned from the list iterator
31.      for (ListIterator<String> it = a.listIterator(a.size()); it.hasPrevious();) {
32.          System.out.println(it.previous() + ", Previous" + it.previousIndex()
33.              + ", Next: " + it.nextIndex());
34.      }
35.  }
36.  }
▶ }

```

69

## List Interface

- ▶ It displays:
- ▶ [A, B, C, D, E, null, F]
- ▶ The element at 5: null
- ▶ Replace the element at 5 with 'X': null
- ▶ [A, B, C, D, E, X, F]
- ▶ Insert 'Y' before 6!
- ▶ [A, B, C, D, E, X, Y, F]
- ▶ Remove the element at 3: D
- ▶ [A, B, C, E, X, Y, F]
- ▶ ABCEXYF
- ▶ F, Previous5, Next: 6
- ▶ Y, Previous4, Next: 5
- ▶ X, Previous3, Next: 4
- ▶ E, Previous2, Next: 3
- ▶ C, Previous1, Next: 2
- ▶ B, Previous0, Next: 1
- ▶ A, Previous-1, Next: 0

70

## Algorithms on List

java.util.Collections

<code>sort()</code>	sorts a <code>List</code> using a merge sort algorithm, which provides a fast, stable sort. A stable sort is one that does not reorder equal elements.
<code>binarySearch()</code>	searches for an element in an ordered <code>List</code> using the binary search algorithm. Returns the position of value in list, or -1 if value is not found.
<code>copy()</code>	copies the elements from the source <code>List</code> into the destination <code>List</code> .
<code>fill()</code>	assigns each element in a <code>List</code> with the specified value.
<code>reverse()</code>	reverses the order of the elements in the specified <code>List</code> .
<code>rotate()</code>	rotates all the elements in the specified <code>List</code> by the specified distance.
<code>swap()</code>	swaps the elements at the specified positions in a <code>List</code> .
<code>shuffle()</code>	randomly permutes the elements in a <code>List</code> .
<code>replaceAll()</code>	replaces all occurrences of one specified value with another.
<code>indexOfSubList()</code>	searches a <code>List</code> for the first occurrence of sublist. Returns the

71

## List Interface

```

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
public class CollectionsFill {
    public static void main(String args[]) {
        List<String> list = new ArrayList<String>();
        for (int i = 0; i < 10; i++) {
            list.add("");
        }
        Collections.fill(list, "Java");
        System.out.println(list);
    }
}

```

▶ [Java, Java, Java, Java, Java, Java, Java, Java, Java, Java]

72

## List Interface

```

▶ import java.util.Arrays;
▶ import java.util.Collections;
▶ import java.util.List;
▶ public class CollectionSort {
▶     public static void main(String args[]) throws
        Exception {
▶         List list<String> = Arrays.asList("apple", "app",
            "bed");
▶         Collections.sort(list);
▶         System.out.println(list);
▶     }
▶ }

```

▶ [app, apple, bed]

73

## List Interface

- ▶ If the List consists of Date elements, it will be sorted into chronological order.
- ▶ This is because String and Date both implement the **Comparable** interface.
- ▶ Comparable implementations provide a **natural ordering** for a class, which allows objects of that class to be sorted automatically.

74

## List Interface

- ▶ when you need to compare objects of the Java built-in classes or the third-party classes by your idea rather than the built-in comparison logic declared in the classes, it is better to use Comparator.
- ▶ The `Java.util.Comparator` interface is used to establish an external ordering logic for the objects being compared even though they are different classes.

## List Interface

- ▶ Sort Strings by their length

```
class StringLengthComparator implements Comparator<String> {  
    public int compare(String a, String b) {  
        return a.length() - b.length();  
    }  
}
```

## List Interface

```

ArrayList<String> list = new ArrayList<String>();
list.add("software");
list.add("egineering");
list.add("helloworld");
Collections.sort(list);
Iterator iterator = list.iterator();
while (iterator.hasNext()) {
    System.out.print(iterator.next() + " ");
}
ArrayList<String> list = new ArrayList<String>();
list.add("software");
list.add("egineering");
list.add("helloworld");
Collections.sort(list, new StringLengthComparator());
Iterator iterator = list.iterator();
while (iterator.hasNext()) {
    System.out.print(iterator.next() + " ");
}

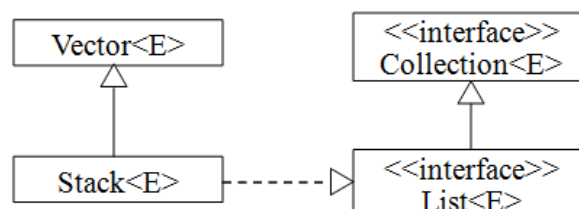
```

Problems @ Javadoc Declaration Cons  
 <terminated> TestTreeSet [Java Application] C:\Prog  
 egineering helloworld software

Problems @ Javadoc Declaration Cons  
 terminated> TestTreeSet [Java Application] C:\Prog  
 software egineering helloworld

## Stack

- ▶ Stack<E> extends Vector<E>
- ▶ The methods provided by Stack<E> are summarized here:
  - push() pushes an object onto the top of this stack.
  - pop() removes the object at the top of this stack and returns that object.
  - empty() tests if this stack is empty;
  - peek() looks at the object at the top of this stack without removing it from the stack.



## Stack

```

> public class StackDemo {
>     public static void main(String[] args) {
>         Car a = new Car("A", 18, 5, 2);
>         Car b = new Car("B", 20, 6, 2);
>         Car c = new Car("C", 17, 7, 2);
>         Car d = new Car("D", 18, 8, 2);
>         Car e = new Car("E", 19, 9, 2);
>
>         Stack<Car> s = new Stack<Car>();
>         System.out.println(s.isEmpty() ? "The hole now is empty. " : "The hole now
not empty.");
>         s.push(a);           //Car a moves in
>         s.push(b);           //Car b moves in
>         s.push(c);           //Car c moves in
>         s.pop();             //Car c moves out
>         s.pop();             //another Car, which is b moves out
>         s.push(d);           //Car d moves in
>         s.push(e);           //Car e moves in
>         System.out.println("The hole now contains: " + s);

```

## Stack

```

>         // Checking the top using the peek() method to know the Car
>         near the entrance to the cave
>         System.out.println("The Car on the top is: " + s.peek());
>         // The element at the top of the stack is at position 1. Position 2
>         is next, then 3, and so on.
>         // If the requested object is not found on the stack, -1 is
>         returned.
>         System.out.println("The Car 'A' is at position: " + s.search(a));
>     }
> }

```



## Stack

```

class Car {
    public Car() {
        this("", 0, 0, 0);
    }

    public Car(String name, int weight, double length, double width) {
        this.name = name;
        this.weight = weight;
        this.length = length;
        this.width = width;
    }

    public String toString() {
        return "[" + name + "," + weight + "," + length + "," + width
+ "," + "]"";
    }

    private String name;
    private int weight;
    private double length, width;
}

```

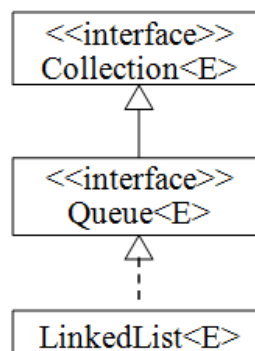
- ▶ The hole now is empty.
- ▶ The hole now is: [[A,18,5.0,2.0,],[D,18,8.0,2.0,],[E,19,9.0,2.0,]]
- ▶ The Car on the top is: [E,19,9.0,2.0,]
- ▶ The Car 'A' is at position: 3

## Queue Interface

```

public interface Queue<E> extends Collection<E> {
    E remove(); //Retrieves and removes the head of this queue.
    E element(); //Retrieves, but does not remove, the
head of this queue.
}

```



## Queue Interface

- ▶ Queue uses the add() method, which inherits from Collection, to insert an element.
- ▶ The add() method implemented by LinkedList appends the specified element to the end of this list.
- ▶ The remove() method implemented by LinkedList removes the head (first element) of this list.
- ▶ The remove() method throws NoSuchElementException when the queue is empty, so does the element() method.

## Queue Interface

```

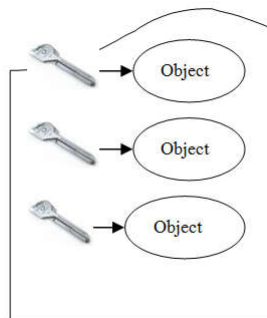
▶ import java.util.LinkedList;
▶ import java.util.Queue;
▶ public class QueueDemo {
▶     public static void main(String[] args) {
▶         Car a = new Car("A", 18, 5, 2);
▶         Car b = new Car("B", 20, 6, 2);
▶         Car c = new Car("C", 17, 7, 2);
▶         Car d = new Car("D", 18, 8, 2);
▶         Car e = new Car("E", 19, 9, 2);
▶         Queue<Car> queue = new LinkedList<Car>();
▶         queue.add(a);
▶         queue.add(b);
▶         queue.add(c);
▶         queue.remove();
▶         queue.remove();
▶         queue.add(d);
▶         queue.add(e);
▶         System.out.println(queue);
▶     }
▶ }

```

▶ It displays:  
 ▶ [[C,17,7.0,2.0,],  
 [D,18,8.0,2.0,],  
 [E,19,9.0,2.0,]]

## Map Interface

- ▶ A Map stores key-value pairs and provides operations to retrieve a value based on the key



85

## Map Interface

```

▶ public interface Map<K,V> {
▶
▶     // Basic operations
▶     V put(K key, V value);           // Stores the value and the key into the Map.
▶     V get(Object key); // Returns the value found in the Map using the specified key.
▶     V remove(Object key); // Returns the value for the specified key and removes the
▶         // key-value pair from the Map
▶     boolean containsKey(Object key); // Returns true if the key is found in the Map
▶     boolean containsValue(Object value); // Returns true if the value is found at least once in the Map.
▶     int size(); // Returns the number of key-value pairs in the Map.
▶     boolean isEmpty(); // Returns true if the Map contains no entries.
▶     void clear(); // Removes all entries from the Map.
▶
▶     // Collection Views
▶     public Set<K> keySet(); // Returns a Set object containing all the keys found in the Map.
▶     public Collection<V> values(); // Returns a Collection object containing all the values found in the Map.
▶     public Set<Map.Entry<K,V>> entrySet(); // Returns a Set object containing all the key-value pairs found in
    the Map.
▶
▶     // Interface for entrySet elements
▶     public interface Entry {
▶         K getKey(); // Returns the key of this Entry.
▶         V getValue(); // Returns the value of this Entry.
▶         V setValue(V value); // Replaces the value of this Entry with the parameter.
▶     }
▶ }

```

86

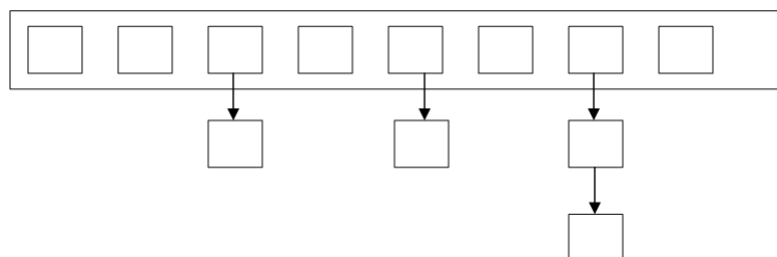
## Map Interface

- ▶ The Java collections framework contains two general-purpose Map implementations:
  - HashMap
  - TreeMap.
- ▶ Any Object used as a key in a HashMap must implement the hashCode( ) and equals( ) methods.

87

## Map Interface

- ▶ HashMap utilizes an array as the storage structure



88

## Map Interface

```

> import java.util.*;
> public class HashMapDemo {
>     public static void main(String[] args) {
>         Car a = new Car("A", 18, 5, 2);
>         Car b = new Car("B", 20, 6, 2);
>         Car c = new Car("C", 17, 7, 2);
>         Car d = new Car("D", 18, 8, 2);
>         Car e = new Car("E", 19, 9, 2);
>         Car f = new Car("F", 20, 10, 2);
>         Car ff = new Car("F", 21, 11, 2);
>
>         Map<String, Car> map = new HashMap<String, Car>();
>
>         map.put("A", a);
>         map.put("B", b);
>         map.put("C", c);
>         map.put("D", d);
>         map.put("E", e);
>         map.put("F", f);
>         map.put("F", ff);

```

89

## Map Interface

```

> //Displaying Contents: public String toString()
>     System.out.println(map);
>
>     // If the key removed is present, the key-value pair will be removed and the
>     // value object will be returned.
>     // If the object removed is not present in the map, null will be returned.
>     Car s3 = map.remove("C");
>     System.out.println(s3 + " has been removed.");
>
>     //Fetching Keys and Values
>     Car s4 = map.get("D");
>
>     //Finding an element
>     System.out.println(map.containsKey("C") ? "C is there." : "C is not there.");
>     System.out.println(map.containsValue("D") ? "D is there." : "D is not there.");

```

90

## Map Interface

```

▶ System.out.println("The number of key-value mappings is " + map.size());
▶ Set<String> s = map.keySet();
▶ for (String k : s) {
▶     Car v = map.get(k);
▶     System.out.print(k + "->" + v + ", ");
▶ }
▶ System.out.println();
▶
▶ //Removal of all elements from a map
▶ map.clear();
▶ System.out.println(map.isEmpty() ? "B is empty." : "B is not empty.");
▶ }
▶ }

```

91

## Map Interface

- ▶ {D=[D,18,8.0,2.0,], A=[A,18,5.0,2.0,], F=[F,21,11.0,2.0,], C=[C,17,7.0,2.0,], B=[B,20,6.0,2.0,], E=[E,19,9.0,2.0,]}
- ▶ [C,17,7.0,2.0,] has been removed.
- ▶ C is not there.
- ▶ D is not there.
- ▶ The number of key-value mappings is 5
- ▶ D->[D,18,8.0,2.0,], A->[A,18,5.0,2.0,], F->[F,21,11.0,2.0,], B->[B,20,6.0,2.0,], E->[E,19,9.0,2.0,],
- ▶ B is empty.

92

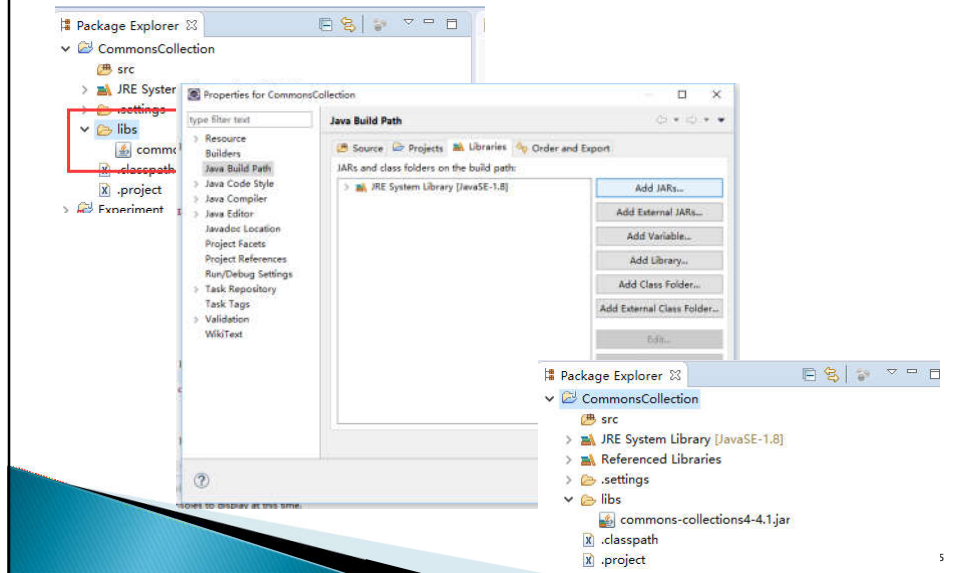
## Third-party Java Library

The screenshot displays the Apache Commons website for the `CollectionUtils` class. The page is titled "Class CollectionUtils" and shows the package `org.apache.commons.collections4`. It includes a "Field Summary" table with one field, `EMPTY_COLLECTION`, and a "Method Summary" table with four methods: `addAll(Collection, Collection)`, `addAll(Collection, Object...)`, `addAll(Collection, Iterable)`, and `addAll(Collection, Collection, Collection)`. The page also includes a "Source" link and a "Download" link.

## Commons Collections

- ▶ Commons-Collections seek to build upon the JDK classes by providing new interfaces, implementations and utilities.
  - **Bag interface** for collections that have a number of copies of each object
  - **BidiMap** interface for maps that can be looked up from value to key as well and key to value
  - **MapIterator** interface to provide simple and quick iteration over maps
  - .....
  - New algorithms for collections

## Commons Collections



## Bag interface

- Defines a collection that counts the number of times an object appears in the collection.

```
import java.util.Set;

import org.apache.commons.collections4.Bag;
import org.apache.commons.collections4.bag.TreeBag;

public class BagTest {

    public static void main(String[] args) {
        String str = "this is a cat and that is a mice whe";
        String[] strArray = str.split(" ");
        Bag<String> bag = new TreeBag<String>();
        for (String temp: strArray) {
            bag.add(temp);
        }
        System.out.println("Counting");
        Set<String> keys = bag.uniqueSet();
        for (String letter: keys) {
            System.out.println(letter + "-->" + bag.getCount(letter));
        }
    }
}
```

```
Problems @ Javadoc
<terminated> BagTest [Java Ap
====Counting====
a-->2
and-->1
cat-->1
food-->1
is-->3
mice-->1
that-->1
the-->1
this-->1
where-->1
```



## BidiMap Interface

- Defines a map that allows bidirectional lookup between key and values.

```
import org.apache.commons.collections4.OrderedBidiMap;
import org.apache.commons.collections4.bidimap.TreeBidiMap;

public class BagTest {

    public static void main(String[] args) {
        TreeBidiMap<String, String> bidi = new TreeBidiMap<String, String>();
        bidi.put("SIX", "6");
        System.out.println(bidi);
        bidi.get("SIX"); // returns "6"
        bidi.getKey("6"); // returns "SIX"
        //bidi.removeValue("6"); // removes the mapping
        OrderedBidiMap<String, String> inverse = bidi.inverseBidiMap(); //
        returns a map with keys and values swapped
        System.out.println(inverse);
    }
}
```

97

## CollectionUtils

- org.apache.commons.collections4.CollectionUtils
- Provides utility methods and decorators for Collection instances.

```
public class CollectionUtilsTest {
    public static void main(String[] args) {
        String[] arrayA = new String[] { "A", "B", "C", "D", "E", "F" };
        String[] arrayB = new String[] { "B", "D", "F", "G", "H", "K" };
        List<String> listA = Arrays.asList(arrayA);
        List<String> listB = Arrays.asList(arrayB);
        System.out.println(CollectionUtils.union(listA, listB));
        //[A, B, C, D, E, F, G, H, K]
        System.out.println(CollectionUtils.intersection(listA, listB));
        //[B, D, F]
        System.out.println(CollectionUtils.disjunction(listA, listB));
        //[A, C, E, G, H, K]
        System.out.println(CollectionUtils.subtract(listA, listB));
        //[A, C, E]
        System.out.println(CollectionUtils.isEmpty(null));
        //true
        System.out.println(CollectionUtils.isEmpty(listA));
        //true
    }
}
```

98