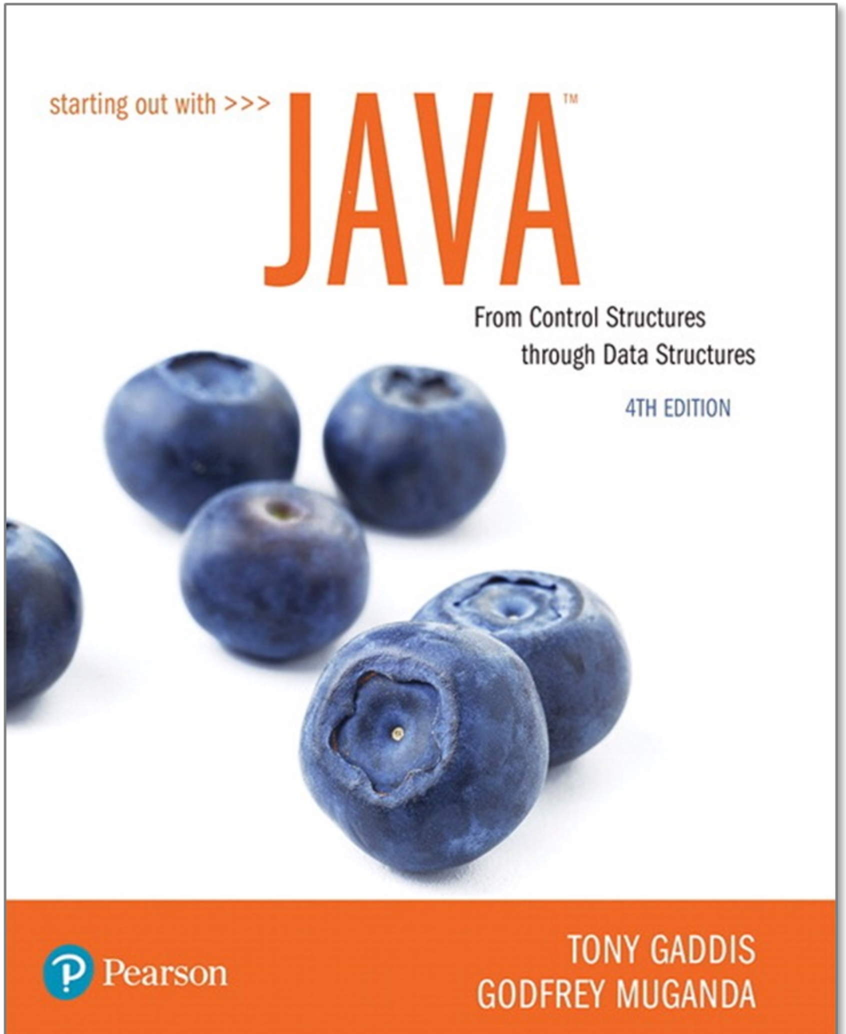
STARTING OUT WITH JAVA™

4th Edition



Chapter 18

Collections and
the Stream API

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Chapter Topics

- Introduction to the Java collections Framework
- Lists
- Sets
- Maps
- The Collections Class
- Functional Interfaces
- The Stream API

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

The Java Collection Framework

The *Java Collections Framework* is a library of classes and interfaces for working with collections of objects.

A *collection* is an object which can store other objects, called *elements*. Collections provide methods for adding and removing elements, and for searching for a particular element within the collection.

Pearson Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

The Main Types of Collections

- Lists
- Sets
- Maps

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Lists

Lists: List type collections assign an integer (called an *index*) to each element stored.

Indices of elements are 0 for the element at the beginning of the list, 1 for the next element, and so on.

Lists permit duplicate elements, which are distinguished by their position in the list.

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Sets

Set: a collection with no notion of position within the collection for stored elements. Sets do not permit duplicate elements.

Pearson Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Maps

A *map* is a collection of pairs of objects:

1. A *value*: this is the object to be stored.
2. A *key*: this is another object associated with the value, and which can be used to quickly find the value within the collection.

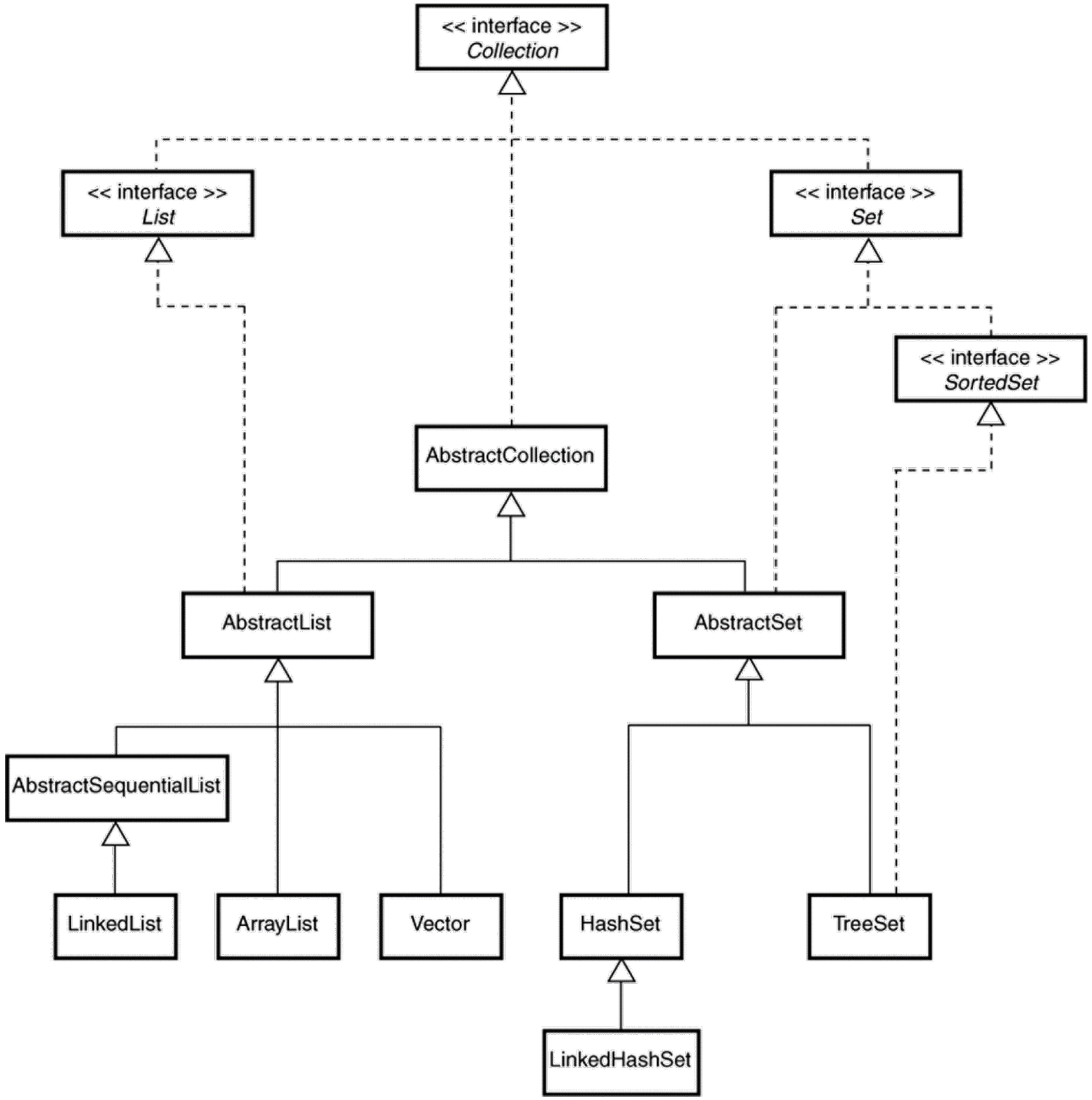
A map is really a set of keys, with each key having a value attached to it.

Maps do not allow duplicate keys.

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Part of the JCF Hierarchy



Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

The Collection Interface

- Lists and Sets are similar in many ways.
- The `Collection` Interface describes the operations that are common to both.
- Maps are fundamentally different from Lists and Sets and are described by a different interface.

Pearson Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Some Methods in the 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.

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Iterators

An *iterator* is an object that is associated with a collection. The iterator provides methods for fetching the elements of the collection, one at a time, in some order.

An iterators has a method for removing from the collection the last item fetched.

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

The Iterator Interface

Iterators implement the `Iterator` interface. This interface specifies the following methods:

hasNext() : boolean

next() : E

remove() : void

The `remove()` method is optional, so not all iterators have it.

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Methods of the Iterator Interface

Method	Description
hasNext() : boolean	Returns true if there is at least one more element from the collection that can be returned, false otherwise.
next() : E	Returns the next element from the collection.
remove() : void	Removes from the collection the element returned by the last call to next(). This method can be called at least one time for each call to next().

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Iterator Example

- This program uses an iterator to fetch and print all elements in a list of strings

```
List<String> names = new ArrayList<>();
names.add("Anna");
names.add("Bob");
names.add("Carlos");

// Get the iterator
Iterator<String>  it = names.iterator();
// Do the iterator thing
while (it.hasNext())
{
    String s = it.next();
    System.out.printf("%s\n", s);
}
```

Pearson Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

The Iterator remove() method

- The `remove()` method removes the element returned by the last call to next().

- The `remove()` method can be called at most one time for each call to next().

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Example of Iterator `remove()`

```
List<String> names = new ArrayList<>();
names.add("Anna");
names.add("Bob");
names.add("Carlos");

// Get iterator
Iterator<String>  it = names.iterator();

// Use iterator to remove second element Bob
it.next();    // return Anna
it.next();    // return Bob
it.remove();  // remove Bob

System.out.println(names);   // prints: [Anna, Carlos]
```

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

The Enhanced For Loop

This manner of using an iterator to access elements of a collection shown below is so common, there is a shorthand for it that is called the enhanced for loop. Instead of writing

```
// Get the iterator
Iterator<String>  it = names.iterator();
// Do the iterator thing
while (it.hasNext())
{
    String s = it.next();
    System.out.printf("%s\n", s);
 }
```

You write:

```
// Enhanced for loop
for (String s : names)
{
     System.out.printf("%s\n", s);
}
```

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Functional Interfaces (1 of 2)

A *functional interface* is an interface that contains a single abstract method

The type of a functional interface is characterized by the parameter type list and return type of the abstract method in the interface

A notation involving a arrow is used to describe the type of a functional notation, for example, the notation

$(S, T) \rightarrow U$ denotes the type of an abstract method taking two parameters of type S and T, in that order, and returning a value of type U

Pearson Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Functional Interfaces (2 of 2)

The java.util.function package defines generic functional interfaces for commonly used types:

Predicate<T> is the type *T -> boolean*
Function<T, R> is the type *T -> R*
Supplier<T> is the type () -> *T*
Consumer<T> is the type *T -> void*

The Collection interface and the Stream API have many methods that take arguments that are functional interfaces

Pearson Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

The `forEach()` Method

The `Collection` method

```
void forEach(Consumer<? super E> action)
```

applies an action to each element of a collection. The action is usually specified by a lambda expression.

The following code prints the length of every string in a list of strings:

```
names.forEach(
            s -> {System.out.printf("%d\n", s.length());}
            );
```

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

The `removeIf()` Method

The `Collection` method

```
void removeIf(Predicate<? super E> filter)
```

removes from the collection all elements on which the filter predicate returns true.

The following code removes from the collection every string that starts with the letter ‘B’.

```
names.removeIf(s -> s.charAt(0) == 'B' );
```

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Lists

Pearson Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

The List Interface

The `List` interface extends the `Collection` interface by adding operations that are specific to the position-based, index-oriented nature of a list.

Pearson
Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

List Interface Methods

The methods in the `List` interface describe operations for adding elements and removing elements from the list based on the index of the element.

There are also methods for determining the index of an element in the list when the value of an element is known.

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

The 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.

Pearson Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

ArrayList

`ArrayList` is an array-based list.

Internally, it uses an array to store its elements: whenever the array gets full, a new, bigger array is created, and the elements are copied to the new array.

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

AbstractSequentialList and LinkedList

Array-based lists have high overhead when elements are being inserted into the list, or removed from the list, at positions that are not at the end of the list.

`LinkedList` is a concrete class that stores elements in a way that eliminates the high overhead of adding to, and removing from positions in the middle of the list.

`LinkedList` extends `AbstractSequentialList`, which in turn, extends `AbstractList`.

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Using the Concrete List Classes

- The concrete classes `ArrayList`, and `LinkedList` work in similar ways, but have different performance characteristics.
- Because they all implement the `List` interface, you can use `List` interface references to instantiate and refer to the different concrete classes.
- Using a `List` interface instead of the concrete class reference allows you to later switch to a different concrete class to get better performance.

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Example: ArrayList

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

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

An Example: LinkedList

Because we used a `List` reference to refer to the concrete class objects, we can easily switch from an `ArrayList` to a `LinkedList` : the only change is in the class used to instantiate the collection.

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Example: LinkedList

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

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

`ListIterator`

The `ListIterator` extends `Iterator` by adding methods for moving backward through the list (in addition to the methods for moving forward that are provided by `Iterator`)

```
hasPrevious() : boolean
previous() : E
```

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Some `ListIterator` Methods

Method	Description
add(el:E):void	Adds el to the list at the position just before the element that will be returned by the next call to the next() method.
hasPrevious():boolean	Returns true if a call to the previous() method will return an element, false if a call to previous() will throw an exception because there is no previous element.
nextIndex():int	Returns the index of the element that would be returned by a call to next(), or the size of the list if there is no such element.
previous():E	Returns the previous element in the list. If the iterator is at the beginning of the list, it throws NoSuchElementException.
previousIndex():int	Returns the index of the element that would be returned by a call to previous(), or -1.
set(el:E):void	Replaces the element returned by the last call to next() or previous() with a new element el.

Pearson Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Iterator Positions

Think of an iterator as having a *cursor position* that is initially just before the element that will be returned by the first call to `next()`.

A call to `next()` puts the cursor just after the element returned, and just before the element that will be returned by the next call to `next()`.

At any time, in a `ListIterator`, the cursor is in between two list elements: A call to `previous()` will skip backward and return the element just skipped, a call to `next()` will skip forward and return the element just skipped.

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Iterator and ListIterator Exceptions

A call to `previous()` throws `NoSuchElementException` when there is no element that can be skipped in a backward move.

A call to `next()` throws `NoSuchElementException` when there is no element that can be skipped in a forward move.

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Example Use of a ListIterator

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

      // Add to arrayList using a ListIterator
      ListIterator<String> it = nameList.listIterator();
      for (int k = 0; k < names.length; k++)
          it.add(names[k]);

      // Get a new  ListIterator for printing
      it = nameList.listIterator();
      while (it.hasNext())
          System.out.println(it.next());
   }
```

Pearson Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Sets

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

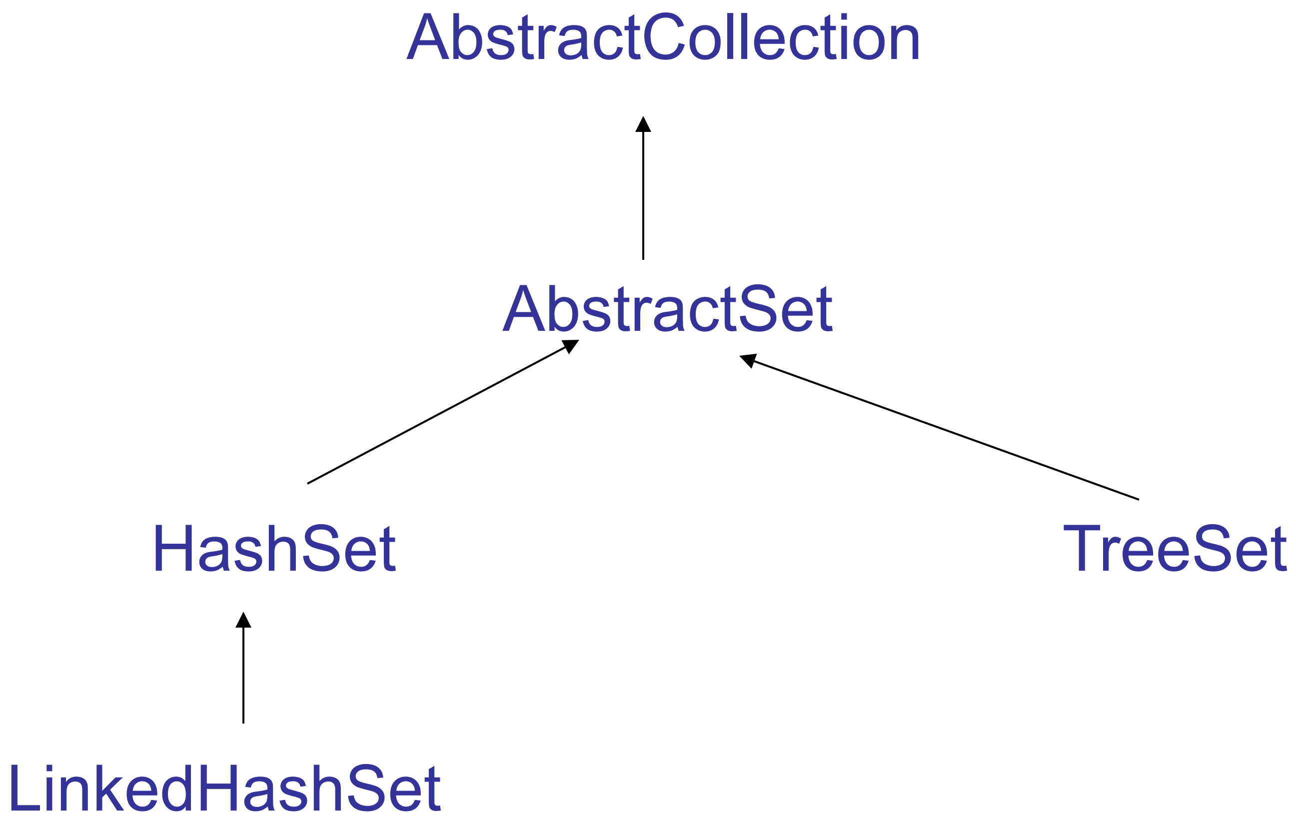
Sets

Sets are collections that store elements, but have no notion of a position of an element within the collection.

The distinguishing feature of a set as a collection is that it does not allow duplicates.

Pearson Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

The Set Part of the JCF Hierarchy



Pearson
Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

The Set Part of the JCF

`AbstractSet` implements the `Set` Interface.

`TreeSet` implements the `SortedSet` interface, which has methods for working with elements that have an order that allows them to be sorted according to their value.

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

HashSet

- `HashSet` containers store elements according to a hash code.
- A `hash code` of an element is an integer computed from the value of the element that can be used to help identify the element.
- The procedure used to compute the hash code of an element is called the `hashing function` or the `hashing algorithm`.

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Examples of Hashing Functions

- For `Integer` objects, you can use the integer value of the object (or its absolute value).
- For `Character` objects, you can use the UNICODE value for the character.
- For `String` objects, you can use a function that takes into account the UNICODE values of the characters that make up the string, as well as the position occupied by each character.

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

A Simplistic Hashing Function

A very simple (but not very good) hashing function for strings might assign to each string the UNICODE value of its first character.

Note that all strings with the same first character are assigned the same hash code.

When two distinct objects have the same hash code, we say that we have a *collision.*

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Implementation of a HashSet

- A `HashSet` can be regarded as a collection of "buckets."
- Each bucket corresponds to a hash code, and stores all objects in the set that have that particular hash code.
- Some buckets will have just one element, whereas other buckets may have many.
- A good hashing scheme should distribute elements among the buckets so that all buckets have approximately the same number of elements.

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Implementation of a HashSet

The `HashSet` is a collection of buckets, and each bucket is a collection of elements.

The collection of buckets is actually a list of buckets, perhaps an `ArrayList`.

Each bucket may also be a list of elements, usually a linked list.

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

How a HashSet Works

- To add an element *X*, the hash code for *X* is used (as an index) to locate the appropriate bucket. *X* is then added to the list for that bucket. If *X* is already in the bucket (The test is done using the `equals` method), then it is not added.
- To remove an item *X*, the hash code for *X* is computed. The corresponding bucket is then searched for *X*, and if *X* is found, it is removed.

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Efficiency of HashSet Operations

Given an item *X*, computing the hash code for *X* and locating the corresponding bucket can be done very fast.

The time to search for, or remove *X* from the bucket depends on how many elements are stored in the bucket.

More collisions mean more elements in some buckets, so we try to find a hashing scheme that minimizes collisions.

Pearson Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

HashSet Performance Considerations

To have good performance with a `HashSet`:

1. Have enough buckets: fewer buckets means more collisions.
2. Have a good hashing function that spreads elements evenly among the buckets. This keeps the number of elements in each bucket small.

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

HashSet Capacity and Load Factor

- The *load factor* of a `HashSet` is the fraction of buckets that must be occupied before the number of buckets is increased.
- The number of buckets in a `HashSet` is called its capacity.

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Some HashSet Constructors

HashSet()	Creates an empty HashSet object with a default initial capacity of 16 and load factor of 0.75.
HashSet(int initCapacity, float loadFactor)	Creates an empty HashSet object with the specified initial capacity and load factor.
HashSet(int initCapacity)	Creates an empty HashSet object with the specified initial capacity and a load factor of 0.75.

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

The hashCode() Method

The Java `Object` class defines a method for computing hash codes

```
int hashCode()
```

This method should be overriden in any class whose instances will be stored in a `HashSet`.

The `Object` class's `hashCode()` method returns a value based on the memory address of the object.

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Overriding the `hashCode()` Method

Observe these guidelines:

1. Objects that are equal according to their `equals` method should be assigned the same hash code.
2. Because of 1), whenever you override a class's `equals()` method, you should also override `hashCode()`.
3. Try to minimize collisions.

Pearson
Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

A `Car` Class for use with a `HashSet`

Note that the `Car` class should override both `equals()` and `hashCode()`.

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

A Car Class for Use With a HashSet

```
// equals() and hashC0de() methods depend on vin only
class Car
{
    String vin,  description;
    public boolean equals(Object other)
    {
        if (!(other instanceof Car))
             return false;
        else
             return  vin.equalsIgnoreCase(((Car)other).vin);
    }

   public int hashCode() { return vin.hashCode();}

   public Car(String v, String d)
      {
          vin = v; description = d;
      }
   public String toString()
     {  return vin + " " + description;  }
}
```

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Use of the Car Class with a HashSet

```
public  static void main(String [ ] args)
   {
      Set<Car> carSet = new HashSet<Car>();
      Car [ ] myRides =
             {
               new Car("TJ1", "Toyota"),
               new Car("GM1", "Corvette"),
               new Car("TJ1", "Toyota Corolla")
             };
      // Add the cars to the HashSet
      for (Car c : myRides)
          carSet.add(c);

      // Print the list using an Iterator
      Iterator it = carSet.iterator();
      while (it.hasNext())
          System.out.println(it.next());
   }
```

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

HashSet<Car> Program Output

```
GM1 Corvette

TJ1 Toyota
```

Note:

- The iterator does not return items in the order added to the `HashSet`.
- The entry of the `Toyota Corolla` is rejected because it is equal to an entry already stored (same vin).

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

LinkedHashSet

A `LinkedHashSet` is just a `HashSet` that keeps track of the order in which elements are added using an auxiliary linked list.

Pearson
Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

TreeSet

A `TreeSet` stores elements based on a natural order defined on those elements.

The natural order is based on the values of the objects being stored .

By internally organizing the storage of its elements according to this order, a `TreeSet` allows fast search for any element in the collection.

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Order

An *order* on a set of objects specifies for any two objects *x* and *y*, exactly one of the following:

x is less than *y*

x is equal to *y*

x is greater than *y*

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Examples of Natural Orders

Some classes have a "natural" order for their objects:

- `Integer`, `Float`, `Double` etc has the obvious concept of natural order which tells when one number is less than another.
- Alphabetic order is the natural order for `String` class objects.

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

The Comparable Interface

In Java, a class can define its natural order by implementing the `Comparable` interface:

```
public interface Comparable<T>
{
int compareTo(T other);
}
```

The `compareTo` method returns a negative value, or zero, or a positive value, to indicate that the calling object is less than, equal to, or greater than the other object.

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Using a `TreeSet` with `Comparable` Elements

1. Make sure the class of your objects implements `Comparable`.
2. Create an instance of `TreeSet` specialized for your class

```
Set<String> mySet = new TreeSet<>();
```

3. Add elements.
4. Retrieve elements using an iterator. The iterator will return elements in sorted order.

Pearson Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Sorting Strings Using a TreeSet

```
import java.util.*;
public class Test
{
   public  static void main(String [ ] args)
    {
       // Create TreeSet
       Set<String> mySet = new TreeSet<String>();
       // Add Strings
       mySet.add("Alan");
       mySet.add("Carol");
       mySet.add("Bob");
       // Get Iterator
       Iterator it = mySet.iterator();
       while (it.hasNext())
       {
           System.out.println(it.next());
       }
    }
}
```

Pearson Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

The SortedSet Interface

`TreeSet` implements the `SortedSet` interface.

`SortedSet` methods allow access to the least and greatest elements in the collection.

`SortedSet` methods allow various views of the collection, for example, the set of all elements greater than a given element, or less than a given element.

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Comparators

A *comparator* for a class is an object that can impose an order on objects of that class.

Comparators are instances of the `Comparator` interface.

`Comparator` is different from the `Comparable` interface, which allows a class to impose an order on its own objects.

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

The Comparator Interface

```
Interface Comparator <T>
{
   int compare(T obj1, T obj2);
   boolean equals(Object o);
}
```

The `compare(x,y)` method returns a negative value, or zero, or a positive value, according to whether x is less than, equal to, or greater than y.

The `equals` method is used to compare one comparator object to another. It does not have to be implemented if the equals inherited from `Object` is adequate.

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Using TreeSet with a Comparator

A `TreeSet` that stores objects of a class that does not implement `Comparable` must use a comparator to order its elements.

The comparator is specified as an argument to the `TreeSet` constructor.

A comparator is used to make a `TreeSet` order its elements differently from their natural order.

Pearson
Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

A Comparator for Ordering Strings in Reverse Alphabetic Order

```
import java.util.*;
class RevStrComparator implements Comparator<String>
{
    public int compare(String s1, String s2)
    {
        return  - s1.compareTo(s2); // Note the negation
    }
}
```

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Using a TreeSet to Sort Strings in Reverse Alphabetic Order

```
public class Test
{
   public  static void main(String [ ] args)
    {  // Create Comparator
       RevStrComparator comp = new RevStrComparator();
       Set<String> mySet = new TreeSet<String>(comp);
       // Add strings
       mySet.add("Alan");
       mySet.add("Carol");
       mySet.add("Bob");
       // Get Iterator
       Iterator it = mySet.iterator();
       while (it.hasNext())
       {
           System.out.println(it.next());
       }
    }
}
```

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Maps (1 of 2)

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Maps (2 of 2)

A *map* is a collection whose elements have two parts: a *key* and a *value*.

The combination of a key and a value is called a *mapping*.

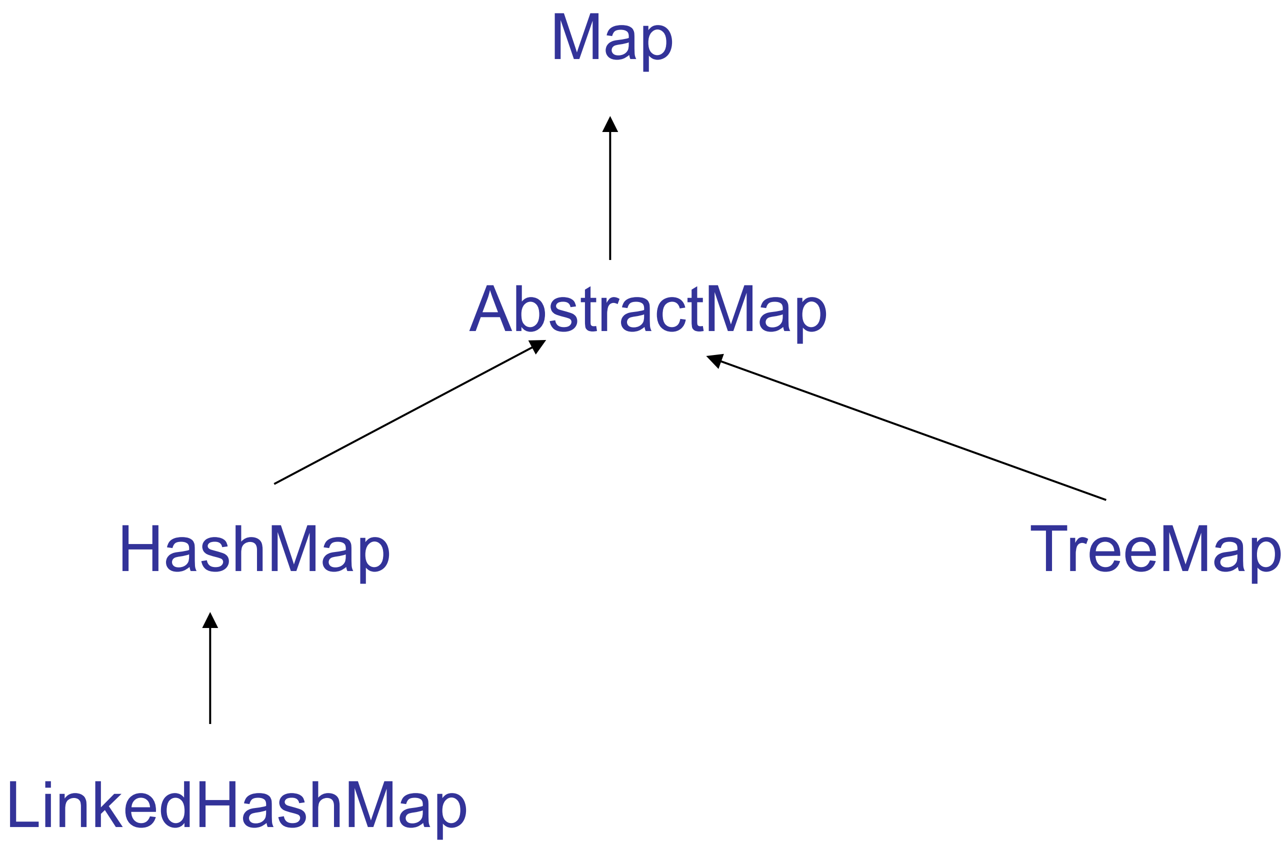
The map stores the mappings based on the key part of the mapping, in a way similar to how a `Set` collection stores its elements.

The map uses *keys* to quickly locate associated *values*.

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

The Map Part of the JCF Hierarchy



Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

The `Map` Interface

`Map` is a generic interface `Map<K, V>`

`Map` specifies two type parameters, `K` for the key, and `V` for the value part of the mapping.

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Some Methods of the `Map` Interface (1 of 2)

clear() : void	Removes all elements from the map.
containsValue(value: Object):boolean	Returns true if the map contains a mapping with the given value.
containsKey(key : Object) : boolean	Returns true if the map contains a mapping with the given key.
get(key : Object) : V	Returns the value associated with the specified key, or returns null if there is no such value.
isEmpty() : boolean	Returns true if the key contains no mappings.
keySet() : Set<K>	Returns the set of all keys stored in the map.

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Some Methods of the `Map` Interface (2 of 2)

put(key : K, value : V) : V	Adds a mapping that associates V with K, and returns the value previously associated with K. Returns null if there was no value associated with K.
remove(key : Object) : V	Removes the mapping associated with the given key from the map, and returns the associated value. If there is not such mapping, returns null.
size() : int	Returns the number of mappings in the map.
values() : Collection<V>	Returns a collection consisting of all values stored in the map.

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Concrete `Map` Classes

Maps store keys with attached values. The keys are stored as sets.

- `HashMap` stores keys according to their hash codes, just like `HashSet` stores its elements.
- `LinkedHashMap` is a `HashMap` that can iterate over the keys in insertion order (order in which mappings were inserted) or in access order (order of last access).
- `TreeMap` stores mappings according to the natural order of the keys, or according to an order specified by a `Comparator`.

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

The Stream API

- A *stream* is an object that permits a pipeline of operations to be performed on a sequence of elements drawn from a source
- A *pipeline* is sequence of operations where the output of one operation becomes the input to the next operation in the sequence
- The *source* may be a collection, an array, or any object that can produce elements on demand

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Pipeline Operations

- An *intermediate* operation is applied to a stream and produces another stream– i.e. it transforms a stream into another stream.

- A *terminal* operation is applied to a stream and produces a non-stream result.

- A terminal operation is also called a *reduction.*

Pearson Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Types of Streams

- A stream of a reference type is created as an instance of the generic stream interface `Stream<T>`.

- A stream of the primitive type `int` is created as an instance of the stream interface `IntStream`.

- The stream interfaces `DoubleStream` and `LongStream` represent streams of the primitive types double and long.

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

A Stream Terminal Operation

- One of the simplest terminal operations on a stream of elements of type T is

```
void forEach(Consumer<? Super T> action)
```

- This operation is similar to the `Collection <T>` method of the same name: it applies the given action to each element drawn from the stream.

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Creating Streams

- The `Arrays` class has static `stream()` methods that take an array as parameter and return a stream of the same element type as the array:

```
static Stream<T> stream(T[ ] array)

static DoubleStream stream(double [ ] array)
static IntStream stream(int[ ] array)

static LongStream stream(long[ ] array)
```

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Creating a Stream from an Array

The following code creates a `Stream<String>` from an array and uses the `forEach` terminal operation to print the elements of the stream:

```
String [] names = {"Anna", "Bob", "Carlos"};
Stream<String> namesStream = Arrays.stream(names);
namesStream.forEach(s -> {System.out.println(s);});
```

Pearson
Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Creating a Stream from a Collection

The `Collection<E>` interface defines an instance method

```
Stream<E> stream()
```

that returns a stream of elements drawn from the collection.

The following code creates an list from an array, and then obtains a stream from the list:

```
String [] names = {"Anna", "Bob", "Carlos"};
List<String> namesList = Arrays.asList(names);
Stream<String> namesStream = namesList.stream();
namesStream.forEach(s -> {System.out.println(s);});
```

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Stream Reduction Operators

- This operator returns the number of elements in the stream:

```
long count()
```

- This operator applies an action to each element of the stream:

```
void forEach(Consumer<? super T> action)
```

- These operators determine whether none, all, or any elements of a stream match a given predicate:

```
boolean noneMatch(Predicate<? super T> p)
boolean allMatch(Predicate<? super T> p)
boolean anyMatch(Predicate<? super T> p)
```

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Intermediate Stream<T> Operators

- This operator removes duplicate and returns the resulting stream:

```
Stream<T> distinct()
```

- This operator returns a stream consisting of only those elements that satisfy the given predicate:

```
Stream<T> filter (Predicate<? Super T> p)
```

- This operator returns a stream consisting of at most the first n elements of this stream:

```
Stream<T> limit(long n)
```

- This operator returns a stream whose elements are the result of applying a function of type *T -> R*

```
<R>Stream<R> map(Function<T, R> mapFunction)
```

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Intermediate Stream<T> Operators

- This operator returns a stream that results from discarding the first n elements:

```
Stream<T> skip(long n)
```

- This operator returns a stream consisting of the same elements, but sorted according to natural order:

```
Stream<T> sorted()
```

- This operator returns a stream consisting of the same elements, but sorted according to the given comparator:

```
Stream<T> sorted(<? super T> comparator)
```

- This operator returns a stream whose elements are the same as this stream. The given action is called for each element as it flows from this stream to the new.

```
Stream<T> peek(Consumer<? Super T> action)
```

Pearson

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved

Copyright

This work is protected by United States copyright laws and is provided solely for the use of instructions in teaching their courses and assessing student learning. dissemination or sale of any part of this work (including on the World Wide Web) will destroy the integrity of the work and is not permitted. The work and materials from it should never be made available to students except by instructors using the accompanying text in their classes. All recipients of this work are expected to abide by these restrictions and to honor the intended pedagogical purposes and the needs of other instructors who rely on these materials.

Copyright © 2019, 2016, 2012 Pearson Education, Inc. All Rights Reserved