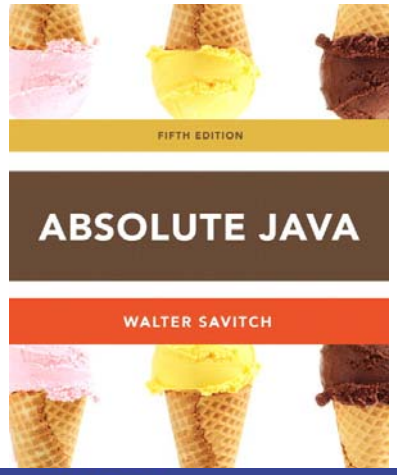


Chapter 16

Collections, Maps and Iterators

Slides prepared by Rose Williams, *Binghamton University*

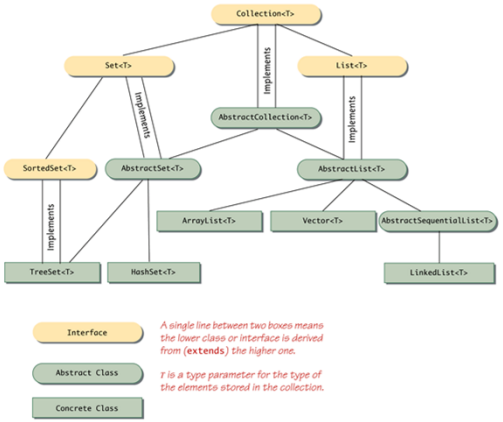
Kenrick Mock, *University of Alaska Anchorage*

Collections

- A Java collection is any class that holds objects and implements the **`Collection`** interface
  - For example, the **`ArrayList<T>`** class is a Java collection class, and implements all the methods in the **`Collection`** interface
  - Collections are used along with *iterators*
- The **`Collection`** interface is the highest level of Java's framework for collection classes
  - All of the collection classes discussed here can be found in package java.util

The `Collection` Landscape

Display 16.1 **The Collection Landscape**



Wildcards

- Classes and interfaces in the collection framework can have parameter type specifications that do not fully specify the type plugged in for the type parameter
  - Because they specify a wide range of argument types, they are known as *wildcards*

```
public void method(String arg1, ArrayList<?> arg2)
```

  - In the above example, the first argument is of type **`String`**, while the second argument can be an **`ArrayList<T>`** with any base type

Wildcards

- A bound can be placed on a wildcard specifying that the type used must be an ancestor type or descendent type of some class or interface
  - The notation `<? extends String>` specifies that the argument plugged in be an object of any descendent class of `String`
  - The notation `<? super String>` specifies that the argument plugged in be an object of any ancestor class of `String`

The `Collection` Framework

- The `Collection<T>` interface describes the basic operations that all collection classes should implement
  - The method headings for these operations are shown on the next several slides
- Since an interface is a type, any method can be defined with a parameter of type `Collection<T>`
  - That parameter can be filled with an argument that is an object of any class in the collection framework

Method Headings in the `Collection<T>` Interface (Part 1 of 10)

Display 16.2 Method Headings in the `Collection<T>` Interface

The `Collection<T>` interface is in the `java.util` package.
All the exception classes mentioned are unchecked exceptions, which means they are not required to be caught in a `catch` block or declared in a `throws` clause.
All the exception classes mentioned are in the package `java.lang` and so do not require any import statement.

CONSTRUCTORS

Although not officially required by the interface, any class that implements the `Collection<T>` interface should have at least two constructors: a no-argument constructor that creates an empty `Collection<T>` object, and a constructor with one parameter of type `Collection<? extends T>` that creates a `Collection<T>` object with the same elements as the constructor argument. The interface does not specify whether the copy produced by the one-argument constructor is a shallow copy or a deep copy of its argument.

```
boolean isEmpty()
```

Returns `true` if the calling object is empty; otherwise returns `false`.

(continued)

Method Headings in the `Collection<T>` Interface (Part 2 of 10)

Display 16.2 Method Headings in the `Collection<T>` Interface

```
public boolean contains(Object target)
```

Returns `true` if the calling object contains at least one instance of `target`. Uses `target.equals` to determine if `target` is in the calling object.
Throws a `ClassCastException` if the type of `target` is incompatible with the calling object (optional).
Throws a `NullPointerException` if `target` is `null` and the calling object does not support `null` elements (optional).

(continued)

Method Headings in the `Collection<T>` Interface (Part 3 of 10)

Display 16.2 Method Headings in the `Collection<T>` Interface

```
public boolean containsAll(Collection<?> collectionOfTargets)
```

Returns `true` if the calling object contains all of the elements in `collectionOfTargets`. For an `element` in `collectionOfTargets`, this method uses `element.equals` to determine if `element` is in the calling object.
Throws a `ClassCastException` if the types of one or more elements in `collectionOfTargets` are incompatible with the calling object (optional).
Throws a `NullPointerException` if `collectionOfTargets` contains one or more `null` elements and the calling object does not support `null` elements (optional).
Throws a `NullPointerException` if `collectionOfTargets` is `null`.

```
public boolean equals(Object other)
```

This is the `equals` of the collection, not the `equals` of the elements in the collection. Overrides the inherited method `equals`. Although there are no official constraints on `equals` for a collection, it should be defined as we have described in Chapter 7 and also to satisfy the intuitive notion of collections being equal.

(continued)

Method Headings in the `Collection<T>` Interface (Part 4 of 10)

Display 16.2 Method Headings in the `Collection<T>` Interface

```
public int size()
```

Returns the number of elements in the calling object. If the calling object contains more than `Integer.MAX_VALUE` elements, returns `Integer.MAX_VALUE`.

```
Iterator<T> iterator()
```

Returns an iterator for the calling object. (Iterators are discussed in Section 16.2.)

```
public Object[] toArray()
```

Returns an array containing all of the elements in the calling object. If the calling object makes any guarantees as to what order its elements are returned by its iterator, this method must return the elements in the same order.
The array returned should be a new array so that the calling object has no references to the returned array. (You might also want the elements in the array to be clones of the elements in the collection. However, this is apparently not required by the interface, since library classes, such as `Vector<T>`, return arrays that contain references to the elements in the collection.)

(continued)

Method Headings in the `Collection<T>` Interface (Part 5 of 10)

Display 16.2 Method Headings in the `Collection<T>` Interface

```
public <E> E[] toArray(E[] a)
```

Note that the type parameter E is not the same as T. So, E can be any reference type; it need not be the type T in `Collection<T>`. For example, E might be an ancestor type of T.
Returns an array containing all of the elements in the calling object. The argument `a` is used primarily to specify the type of the array returned. The exact details are as follows:
The type of the returned array is that of `a`. If the elements in the calling object fit in the array `a`, then `a` is used to hold the elements of the returned array; otherwise a new array is created with the same type as `a`.
If `a` has more elements than the calling object, the element in `a` immediately following the end of the copied elements is set to `null`.
If the calling object makes any guarantees as to what order its elements are returned by its iterator, this method must return the elements in the same order. (Iterators are discussed in Section 16.2.)
Throws an `ArrayStoreException` if the type of `a` is not an ancestor type of the type of every element in the calling object.
Throws a `NullPointerException` if `a` is `null`.

(continued)

Method Headings in the `Collection<T>` Interface (Part 6 of 10)

Display 16.2 Method Headings in the `Collection<T>` Interface

```
public int hashCode()
```

Returns the hash code value for the calling object. Neither hash codes nor this method are discussed in this book. This entry is only here to make the definition of the `Collection<T>` interface complete. You can safely ignore this entry until you go on to study hash codes in a more advanced book. In the meantime, if you need to implement this method, have the method throw an `UnsupportedOperationException`.

OPTIONAL METHODS

The following methods are optional, which means they still must be implemented, but the implementation can simply throw an `UnsupportedOperationException` if, for some reason, you do not want to give them a "real" implementation. An `UnsupportedOperationException` is a `RunTimeException` and so is not required to be caught or declared in a `throws` clause.

(continued)

Method Headings in the `Collection<T>` Interface (Part 7 of 10)

Display 16.2 Method Headings in the `Collection<T>` Interface

`public boolean add(T element)` *(Optional)*

Ensures that the calling object contains the specified `element`. Returns `true` if the calling object changed as a result of the call. Returns `false` if the calling object does not permit duplicates and already contains `element`; also returns `false` if the calling object does not change for any other reason.
Throws an `UnsupportedOperationException` if this method is not supported by the class that implements this interface.
Throws a `ClassCastException` if the class of `element` prevents it from being added to the calling object.
Throws a `NullPointerException` if `element` is `null` and the calling object does not support `null` elements.
Throws an `IllegalArgumentException` if some other aspect of `element` prevents it from being added to the calling object.

(continued)

Method Headings in the `Collection<T>` Interface (Part 8 of 10)

Display 16.2 Method Headings in the `Collection<T>` Interface

`public boolean addAll(Collection<? extends T> collectionToAdd)` *(Optional)*

Ensures that the calling object contains all the elements in `collectionToAdd`. Returns `true` if the calling object changed as a result of the call; returns `false` otherwise. If the calling object changes during this operation, its behavior is unspecified; in particular, its behavior is unspecified if `collectionToAdd` is the calling object.
Throws an `UnsupportedOperationException` if this method is not supported by the class that implements this interface.
Throws a `ClassCastException` if the class of an element of `collectionToAdd` prevents it from being added to the calling object.
Throws a `NullPointerException` if `collectionToAdd` contains one or more `null` elements and the calling object does not support `null` elements, or if `collectionToAdd` is `null`.
Throws an `IllegalArgumentException` if some aspect of an element of `collectionToAdd` prevents it from being added to the calling object.

(continued)

Method Headings in the `Collection<T>` Interface (Part 9 of 10)

Display 16.2 Method Headings in the `Collection<T>` Interface

`public boolean remove(Object element)` *(Optional)*

Removes a single instance of the `element` from the calling object, if it is present. Returns `true` if the calling object contained the `element`; returns `false` otherwise.
Throws an `UnsupportedOperationException` if this method is not supported by the class that implements this interface.
Throws a `ClassCastException` if the type of `element` is incompatible with the calling object (optional).
Throws a `NullPointerException` if `element` is `null` and the calling object does not support `null` elements (optional).

`public boolean removeAll(Collection<?> collectionToRemove)` *(Optional)*

Removes all the calling object's elements that are also contained in `collectionToRemove`. Returns `true` if the calling object was changed; otherwise returns `false`.
Throws an `UnsupportedOperationException` if this method is not supported by the class that implements this interface.
Throws a `ClassCastException` if the types of one or more elements in `collectionToRemove` are incompatible with the calling collection (optional).
Throws a `NullPointerException` if `collectionToRemove` contains one or more `null` elements and the calling object does not support `null` elements (optional).
Throws a `NullPointerException` if `collectionToRemove` is `null`.

(continued)

Method Headings in the `Collection<T>` Interface (Part 10 of 10)

Display 16.2 Method Headings in the `Collection<T>` Interface

`public void clear()` *(Optional)*

Removes all the elements from the calling object.
Throws an `UnsupportedOperationException` if this method is not supported by the class that implements this interface.

`public boolean retainAll(Collection<?> saveElements)` *(Optional)*

Retains only the elements in the calling object that are also contained in the collection `saveElements`. In other words, removes from the calling object all of its elements that are not contained in the collection `saveElements`. Returns `true` if the calling object was changed; otherwise returns `false`.
Throws an `UnsupportedOperationException` if this method is not supported by the class that implements this interface.
Throws a `ClassCastException` if the types of one or more elements in `saveElements` are incompatible with the calling object (optional).
Throws a `NullPointerException` if `saveElements` contains one or more `null` elements and the calling object does not support `null` elements (optional).
Throws a `NullPointerException` if `saveElements` is `null`.

Collection Relationships

- There are a number of different predefined classes that implement the **`Collection<T>`** interface
  - Programmer defined classes can implement it also
- A method written to manipulate a parameter of type **`Collection<T>`** will work for all of these classes, either singly or intermixed
- There are two main interfaces that extend the **`Collection<T>`** interface: The **`Set<T>`** interface and the **`List<T>`** interface

Collection Relationships

- Classes that implement the **`Set<T>`** interface do not allow an element in the class to occur more than once
  - The **`Set<T>`** interface has the same method headings as the **`Collection<T>`** interface, but in some cases the *semantics* (intended meanings) are different
  - Methods that are optional in the **`Collection<T>`** interface are required in the **`Set<T>`** interface

Collection Relationships

- Classes that implement the **`List<T>`** interface have their elements ordered as on a list
  - Elements are indexed starting with zero
  - A class that implements the **`List<T>`** interface allows elements to occur more than once
  - The **`List<T>`** interface has more method headings than the **`Collection<T>`** interface
  - Some of the methods inherited from the **`Collection<T>`** interface have different semantics in the **`List<T>`** interface
  - The **`ArrayList<T>`** class implements the **`List<T>`** interface

Methods in the **`Set<T>`**

- The `Set<T>` interface has the same method headings as the `Collection<T>` interface, but in some cases the semantics are different. For example the `add` methods:

The `Set<T>` interface is in the `java.util` package.
The `Set<T>` interface extends the `Collection<T>` interface and has all the same method headings given in Display 16.2. However, the semantics of the add methods vary as described below.

`public boolean add(T element)` ***`(Optional)`***
If element is not already in the calling object, element is added to the calling object and true is returned. If element is in the calling object, the calling object is unchanged and false is returned.

`public boolean addAll(Collection<? extends T> collectionToAdd)`
`(Optional)`
Ensures that the calling object contains all the elements in `collectionToAdd`. Returns true if the calling object changed as a result of the call; returns false otherwise. Thus, if `collectionToAdd` is a `Set<T>`, then the calling object is changed to the union of itself with `collectionToAdd`.

Methods in the `List<T>` Interface (Part 1 of 16)

The List<T> interface has more method headings than the Collection<T> interface.

Display 16.4 Methods in the `List<T>` Interface

The `List<T>` interface is in the `java.util` package.
The `List<T>` interface extends the `Collection<T>` interface.
All the exception classes mentioned are the kind that are not required to be caught in a `catch` block or declared in a `throws` clause.
All the exception classes mentioned are in the package `java.lang` and so do not require any import statement.

CONSTRUCTORS

Although not officially required by the interface, any class that implements the `List<T>` interface should have at least two constructors: a no-argument constructor that creates an empty `List<T>` object, and a constructor with one parameter of type `Collection<? extends T>` that creates a `List<T>` object with the same elements as the constructor argument. If the argument imposes an ordering on its elements, then the `List<T>` created should preserve this ordering.

`boolean isEmpty()`

Returns `true` if the calling object is empty; otherwise returns `false`.

(continued)

Methods in the `List<T>` Interface (Part 2 of 16)

Display 16.4 Methods in the `List<T>` Interface

`public boolean contains(Object target)`

Returns `true` if the calling object contains at least one instance of `target`. Uses `target.equals` to determine if `target` is in the calling object.
Throws a `ClassCastException` if the type of `target` is incompatible with the calling object (optional).
Throws a `NullPointerException` if `target` is `null` and the calling object does not support `null` elements (optional).

`public boolean containsAll(Collection<?> collectionOfTargets)`

Returns `true` if the calling object contains all of the elements in `collectionOfTargets`. For an element in `collectionOfTargets`, this method uses `element.equals` to determine if `element` is in the calling object. The elements need not be in the same order or have the same multiplicity in `collectionOfTargets` and in the calling object.
Throws a `ClassCastException` if the types of one or more elements in `collectionOfTargets` are incompatible with the calling object (optional).
Throws a `NullPointerException` if `collectionOfTargets` contains one or more `null` elements and the calling object does not support `null` elements (optional).
Throws a `NullPointerException` if `collectionOfTargets` is `null`.

(continued)

Methods in the `List<T>` Interface (Part 3 of 16)

Display 16.4 Methods in the `List<T>` Interface

`public boolean equals(Object other)`

If the argument is a `List<T>`, returns `true` if the calling object and the argument contain exactly the same elements in exactly the same order; otherwise returns `false`. If the argument is not a `List<T>`, `false` is returned.

`public int size()`

Returns the number of elements in the calling object. If the calling object contains more than `Integer.MAX_VALUE` elements, returns `Integer.MAX_VALUE`.

`Iterator<T> iterator()`

Returns an iterator for the calling object. (Iterators are discussed in Section 16.2.)

(continued)

Methods in the `List<T>` Interface (Part 4 of 16)

Display 16.4 Methods in the `List<T>` Interface

`public Object[] toArray()`

Returns an array containing all of the elements in the calling object. The elements in the returned array are in the same order as in the calling object. A new array must be returned so that the calling object has no references to the returned array.

`public <E> E[] toArray(E[] a)`

Note that the type parameter E is not the same as T. So, E can be any reference type; it need not be the type T in `Collection<T>`. For example, E might be an ancestor type of T.
Returns an array containing all of the elements in the calling object. The elements in the returned array are in the same order as in the calling object. The argument `a` is used primarily to specify the type of the array returned. The exact details are described in the table for the `Collection<T>` interface (Display 16.2).
Throws an `ArrayStoreException` if the type of `a` is not an ancestor type of the type of every element in the calling object.
Throws a `NullPointerException` if `a` is `null`.

(continued)

Methods in the `List<T>` Interface (Part 5 of 16)

Display 16.4 **Methods in the `List<T>` Interface**

`public int hashCode()`

Returns the hash code value for the calling object. Neither hash codes nor this method are discussed in this book. This entry is here only to make the definition of the `list` interface complete. You can safely ignore this entry until you go on to study hash codes in a more advanced book. In the meantime, if you need to implement this method, have it throw an `UnsupportedOperationException`.

OPTIONAL METHODS

As with the `Collection<T>` interface, the following methods are optional, which means they still must be implemented, but the implementation can simply throw an `UnsupportedOperationException` if for some reason you do not want to give them a "real" implementation. An `UnsupportedOperationException` is a `RunTimeException` and so is not required to be caught or declared in a `throws` clause.

(continued)

Methods in the `List<T>` Interface (Part 6 of 16)

Display 16.4 **Methods in the `List<T>` Interface**

`public boolean addAll(Collection<? extends T> collectionToAdd)` *(Optional)*

Adds all of the elements in `collectionToAdd` to the end of the calling object's list. The elements are added in the order they are produced by an iterator for `collectionToAdd`.
Throws an `UnsupportedOperationException` if the `addAll` method is not supported by the calling object.
Throws a `ClassCastException` if the class of an element in `collectionToAdd` prevents it from being added to the calling object.
Throws a `NullPointerException` if `collectionToAdd` contains one or more `null` elements and the calling object does not support `null` elements, or if `collectionToAdd` is `null`.
Throws an `IllegalArgumentException` if some aspect of an element in `collectionToAdd` prevents it from being added to the calling object.

`public boolean remove(Object element)` *(Optional)*

Removes the first occurrence of `element` from the calling object's list, if it is present. Returns `true` if the calling object contained the `element`; returns `false` otherwise.
Throws a `ClassCastException` if the type of `element` is incompatible with the calling object (optional).
Throws a `NullPointerException` if `element` is `null` and the calling object does not support `null` elements (optional).
Throws an `UnsupportedOperationException` if the `remove` method is not supported by the calling object.

(continued)

Methods in the `List<T>` Interface (Part 7 of 16)

Display 16.4 **Methods in the `List<T>` Interface**

`public boolean add(T element)` *(Optional)*

Adds `element` to the end of the calling object's list. Normally returns `true`. Returns `false` if the operation failed, but if the operation failed, something is seriously wrong and you will probably get a run-time error anyway.
Throws an `UnsupportedOperationException` if the `add` method is not supported by the calling object.
Throws a `ClassCastException` if the class of `element` prevents it from being added to the calling object.
Throws a `NullPointerException` if `element` is `null` and the calling object does not support `null` elements.
Throws an `IllegalArgumentException` if some aspect of `element` prevents it from being added to the calling object.

(continued)

Methods in the `List<T>` Interface (Part 8 of 16)

Display 16.4 **Methods in the `List<T>` Interface**

`public boolean removeAll(Collection<?> collectionToRemove)` *(Optional)*

Removes all the calling object's elements that are also in `collectionToRemove`. Returns `true` if the calling object was changed; otherwise returns `false`.
Throws an `UnsupportedOperationException` if the `removeAll` method is not supported by the calling object.
Throws a `ClassCastException` if the types of one or more elements in the calling object are incompatible with `collectionToRemove` (optional).
Throws a `NullPointerException` if the calling object contains one or more `null` elements and `collectionToRemove` does not support `null` elements (optional).
Throws a `NullPointerException` if `collectionToRemove` is `null`.

`public void clear()` *(Optional)*

Removes all the elements from the calling object.
Throws an `UnsupportedOperationException` if the `clear` method is not supported by the calling object.

(continued)

Methods in the `List<T>` Interface (Part 9 of 16)

Display 16.4 Methods in the `List<T>` Interface

```
public boolean retainAll(Collection<?> saveElements) (Optional)
```

Retains only the elements in the calling object that are also in the collection `saveElements`. In other words, removes from the calling object all of its elements that are not contained in the collection `saveElements`. Returns `true` if the calling object was changed; otherwise returns `false`.
Throws an `UnsupportedOperationException` if the `retainAll` method is not supported by the calling object.
Throws a `ClassCastException` if the types of one or more elements in the calling object are incompatible with `saveElements` (optional).
Throws a `NullPointerException` if the calling object contains one or more `null` elements and `saveElements` does not support `null` elements (optional).
Throws a `NullPointerException` if the `saveElements` is `null`.

NEW METHOD HEADINGS

The following methods are in the `List<T>` interface but were not in the `Collection<T>` interface. Those that are optional are noted.

(continued)

Methods in the `List<T>` Interface (Part 10 of 16)

Display 16.4 Methods in the `List<T>` Interface

```
public void add(int index, T newElement) (Optional)
```

Inserts `newElement` in the calling object's list at location `index`. The old elements at location `index` and higher are moved to higher indices.
Throws an `IndexOutOfBoundsException` if the `index` is not in the range:

```
0 <= index <= size()
```

Throws an `UnsupportedOperationException` if this `add` method is not supported by the calling object.
Throws a `ClassCastException` if the class of `newElement` prevents it from being added to the calling object.
Throws a `NullPointerException` if `newElement` is `null` and the calling object does not support `null` elements.
Throws an `IllegalArgumentException` if some aspect of `newElement` prevents it from being added to the calling object.

(continued)

Methods in the `List<T>` Interface (Part 11 of 16)

Display 16.4 Methods in the `List<T>` Interface

```
public boolean addAll(int index,
                      Collection<? extends T> collectionToAdd) (Optional)
```

Inserts all of the elements in `collectionToAdd` to the calling object's list starting at location `index`. The old elements at location `index` and higher are moved to higher indices. The elements are added in the order they are produced by an iterator for `collectionToAdd`.
Throws an `IndexOutOfBoundsException` if the `index` is not in the range:

```
0 <= index <= size()
```

Throws an `UnsupportedOperationException` if the `addAll` method is not supported by the calling object.
Throws a `ClassCastException` if the class of one of the elements of `collectionToAdd` prevents it from being added to the calling object.
Throws a `NullPointerException` if `collectionToAdd` contains one or more `null` elements and the calling object does not support `null` elements, or if `collectionToAdd` is `null`.
Throws an `IllegalArgumentException` if some aspect of one of the elements of `collectionToAdd` prevents it from being added to the calling object.

(continued)

Methods in the `List<T>` Interface (Part 12 of 16)

Display 16.4 Methods in the `List<T>` Interface

```
public T get(int index)
```

Returns the object at position `index`.
Throws an `IndexOutOfBoundsException` if the `index` is not in the range:

```
0 <= index < size()
```

```
public T set(int index, T newElement) (Optional)
```

Sets the element at the specified `index` to `newElement`. The element previously at that position is returned.
Throws an `IndexOutOfBoundsException` if the `index` is not in the range:

```
0 <= index < size()
```

Throws an `UnsupportedOperationException` if the `set` method is not supported by the calling object.
Throws a `ClassCastException` if the class of `newElement` prevents it from being added to the calling object.
Throws a `NullPointerException` if `newElement` is `null` and the calling object does not support `null` elements.
Throws an `IllegalArgumentException` if some aspect of `newElement` prevents it from being added to the calling object.

(continued)

Methods in the `List<T>` Interface (Part 13 of 16)

Display 16.4 **Methods in the `List<T>` Interface**

`public T remove(int index)` *(Optional)*

Removes the element at position `index` in the calling object. Shifts any subsequent elements to the left (subtracts one from their indices). Returns the element that was removed from the calling object.
Throws an `UnsupportedOperationException` if the `remove` method is not supported by the calling object.
Throws an `IndexOutOfBoundsException` if `index` does not satisfy:

```
0 <= index < size()
```

(continued)

Methods in the `List<T>` Interface (Part 14 of 16)

Display 16.4 **Methods in the `List<T>` Interface**

`public int indexOf(Object target)`

Returns the index of the first element that is equal to `target`. Uses the method `equals` of the object `target` to test for equality. Returns `-1` if `target` is not found.
Throws a `ClassCastException` if the type of `target` is incompatible with the calling object (optional).
Throws a `NullPointerException` if `target` is `null` and the calling object does not support `null` elements (optional).

`public int lastIndexOf(Object target)`

Returns the index of the last element that is equal to `target`. Uses the method `equals` of the object `target` to test for equality. Returns `-1` if `target` is not found.
Throws a `ClassCastException` if the type of `target` is incompatible with the calling object (optional).
Throws a `NullPointerException` if `target` is `null` and the calling object does not support `null` elements (optional).

(continued)

Methods in the `List<T>` Interface (Part 15 of 16)

Display 16.4 **Methods in the `List<T>` Interface**

`public List<T> subList(int fromIndex, int toIndex)`

Returns a *view* of the elements at locations `fromIndex` to `toIndex` of the calling object; the object at `fromIndex` is included; the object, if any, at `toIndex` is not included. The *view* uses references into the calling object; so, changing the view can change the calling object. The returned object will be of type `List<T>` but need not be of the same type as the calling object. Returns an empty `List<T>` if `fromIndex` equals `toIndex`.
Throws an `IndexOutOfBoundsException` if `fromIndex` and `toIndex` do not satisfy:

```
0 <= fromIndex <= toIndex <= size()
```

(continued)

Methods in the `List<T>` Interface (Part 16 of 16)

Display 16.4 **Methods in the `List<T>` Interface**

`ListIterator<T> listIterator()`

Returns a list iterator for the calling object. (Iterators are discussed in Section 16.2.)

`ListIterator<T> listIterator(int index)`

Returns a list iterator for the calling object starting at `index`. The first element to be returned by the iterator is the one at `index`. (Iterators are discussed in Section 16.2.)
Throws an `IndexOutOfBoundsException` if `index` does not satisfy:

```
0 <= index <= size()
```

Pitfall: Optional Operations

- When an interface lists a method as "optional," it must still be implemented in a class that implements the interface
  - The optional part means that it is permitted to write a method that does not completely implement its intended semantics
  - However, if a trivial implementation is given, then the method body should throw an **`UnsupportedOperationException`**

Tip: Dealing with All Those Exceptions

- The tables of methods for the various collection interfaces and classes indicate that certain exceptions are thrown
  - These are unchecked exceptions, so they are useful for debugging, but need not be declared or caught
- In an existing collection class, they can be viewed as run-time error messages
- In a derived class of some other collection class, most or all of them will be inherited
- In a collection class defined from scratch, if it is to implement a collection interface, then it should throw the exceptions that are specified in the interface

Concrete Collections Classes

- The concrete class **`HashSet<T>`** implements the **`Set<T>`** interface, and can be used if additional methods are not needed
  - The **`HashSet<T>`** class implements all the methods in the **`Set<T>`** interface, and adds only constructors
  - The **`HashSet<T>`** class is implemented using a *hash table*
- The **`ArrayList<T>`** and **`Vector<T>`** classes implement the **`List<T>`** interface, and can be used if additional methods are not needed
  - Both the **`ArrayList<T>`** and **`Vector<T>`** interfaces implement all the methods in the interface **`List<T>`**
  - Either class can be used when a **`List<T>`** with efficient random access to elements is needed

Concrete Collections Classes

- The concrete class **`LinkedList<T>`** is a concrete derived class of the abstract class **`AbstractSequentialList<T>`**
  - When efficient sequential movement through a list is needed, the **`LinkedList<T>`** class should be used
- The interface **`SortedSet<T>`** and the concrete class **`TreeSet<T>`** are designed for implementations of the **`Set<T>`** interface that provide for rapid retrieval of elements
  - The implementation of the class is similar to a binary tree, but with ways to do inserting that keep the tree balanced

Methods in the **HashSet<T>** Class (Part 1 of 2)

Display 16.5 **Methods in the HashSet<T> Class**

The HashSet<T> class is in the `java.util` package.
The HashSet<T> class extends the AbstractSet<T> class and implements the Set<T> interface.
The HashSet<T> class implements all of the methods in the Set<T> interface (Display 16.3). The only other methods in the HashSet<T> class are the constructors. The three constructors that do not involve concepts beyond the scope of this book are given below.
All the exception classes mentioned are the kind that are not required to be caught in a `catch` block or declared in a `throws` clause.
All the exception classes mentioned are in the package `java.lang` and so do not require any import statement.

```
public HashSet()
```

Creates a new, empty set.

(continued)

Methods in the **HashSet<T>** Class (Part 2 of 2)

Display 16.5 **Methods in the HashSet<T> Class**

```
public HashSet(Collection<? extends T> c)
```

Creates a new set that contains all the elements of `c`.
Throws a `NullPointerException` if `c` is `null`.

```
public HashSet(int initialCapacity)
```

Creates a new, empty set with the specified capacity.
Throws an `IllegalArgumentException` if `initialCapacity` is less than zero.

The methods are the same as those described for the Set<T> interface (Display 16.3.)

HashSet<T> Class Demo (1 of 4)

```
1       import java.util.HashSet;
2       import java.util.Iterator;
3       public class HashSetDemo
4       {
5               private static void outputSet(HashSet<String> set)
6               {
7                       Iterator<String> i = set.iterator( );
8                       while (i.hasNext( ))
9                               System.out.print(i.next( ) + " ");
10                      System.out.println();
11              }

12              public static void main(String[] args)
13              {
14                      HashSet<String> round = new HashSet<String>( );
15                      HashSet<String> green = new HashSet<String>( );

16                      // Add some data to each set
17                      round.add("peas");
18                      round.add("ball");
19                      round.add("pie");
20                      round.add("grapes");
```


HashSet<T> Class Demo (2 of 4)

```
25                      System.out.println("Contents of set round: ");
26                      outputSet(round);
27                      System.out.println("\nContents of set green: ");
28                      outputSet(green);

29                      System.out.println("\nball in set 'round'? " +
30                              round.contains("ball"));
31                      System.out.println("ball in set 'green'? " +
32                              green.contains("ball"));

33                      System.out.println("\nball and peas in same set? " +
34                              ((round.contains("ball") &&
35                               (round.contains("peas"))) ||
36                             (green.contains("ball") &&
37                                (green.contains("peas")))));
38                      System.out.println("pie and grass in same set? " +
39                              ((round.contains("pie") &&
40                               (round.contains("grass"))) ||
41                                      (green.contains("pie") &&
42                               (green.contains("grass")))));
```

HashSet<T> Class Demo (3 of 4)

```
43                      // To union two sets we use the addAll method.
44                      HashSet<String> setUnion = new HashSet<String>(round)
45                      round.addAll(green);
46                      System.out.println("\nUnion of green and round:");
47                      outputSet(setUnion);

48                      // To intersect two sets we use the removeAll method.
49                      HashSet<String> setInter = new HashSet<String>(round)
50                      setInter.removeAll(green);
51                      System.out.println("\nIntersection of green and round
52                      outputSet(setInter);
53                      System.out.println();
54              }
55 }
```


HashSet<T> Class Demo (4 of 4)

SAMPLE DIALOGUE

```
Contents of set round:
grapes pie ball peas

Contents of set green:
grass garden hose grapes peas

ball in set round?  true
ball in set green?  false

ball and peas in same set? true
pie and grass in same set? false

Union of green and round:
garden hose grass peas ball pie grapes

Intersection of green and round:
peas grapes
```


Using HashSet with your own Class

- If you intend to use the HashSet<T> class with your own class as the parameterized type T , then your class must override the following methods:
  - public int hashCode();
    - Ideally returns a unique integer for this object
  - public boolean equals(Object obj);
    - Indicates whether or not the reference object is the same as the parameter obj

Methods in the Classes **`ArrayList<T>`** and **`Vector<T>`** (Part 1 of 15)

Display 16.6 **Methods in the Classes `ArrayList<T>` and `Vector<T>`**

The `ArrayList<T>` and `Vector<T>` classes and the `Iterator<T>` and `ListIterator<T>` interfaces are in the `java.util` package.
All the exception classes mentioned are unchecked exceptions, which means they are not required to be caught in a `catch` block or declared in a `throws` clause. (If you have not yet studied exceptions, you can consider the exceptions to be run-time error messages.)
`NoSuchElementException` is in the `java.util` package, which requires an import statement if your code mentions the `NoSuchElementException` class. All the other exception classes mentioned are in the package `java.lang` and so do not require any import statement.
In some situations where we specify throwing an `IndexOutOfBoundsException`, the class `Vector<T>` actually throws an `ArrayIndexOutOfBoundsException`.

(continued)

Methods in the Classes **ArrayList<T>** and **Vector<T>** (Part 2 of 15)

Display 16.6 **Methods in the Classes ArrayList<T> and Vector<T>**

`public ArrayList(Collection<? extends T> c)`

Creates a `ArrayList<T>` that contains all the elements of the collection `c` in the same order as they have in `c`. In other words, the elements have the same index in the `ArrayList<T>` created as they do in `c`. This is not quite a true copy constructor because it does not preserve capacity. The capacity of the created list will be `c.size()`, not `c.capacity`.
The `ArrayList<T>` created is only a shallow copy of the collection argument. The `ArrayList<T>` created contains references to the elements in `c` (not references to clones of the elements in `c`).
Throws a `NullPointerException` if `c` is `null`.

`public Vector(int initialCapacity)`

Creates an empty vector with the specified initial capacity. When the vector needs to increase its capacity, the capacity doubles.
Throws an `IllegalArgumentException` if `initialCapacity` is negative.

`public Vector()`

Creates an empty vector with an initial capacity of `10`. When the vector needs to increase its capacity, the capacity doubles.

(continued)

Methods in the Classes **ArrayList<T>** and **Vector<T>** (Part 3 of 15)

Display 16.6 **Methods in the Classes ArrayList<T> and Vector<T>**

CONSTRUCTORS

`public ArrayList(int initialCapacity)`

Creates an empty `ArrayList<T>` with the specified initial capacity. When the `ArrayList<T>` needs to increase its capacity, the capacity doubles.
Throws an `IllegalArgumentException` if `initialCapacity` is negative.

`public ArrayList()`

Creates an empty `ArrayList<T>` with an initial capacity of `10`. When the `ArrayList<T>` needs to increase its capacity, the capacity doubles.

(continued)

Methods in the Classes **ArrayList<T>** and **Vector<T>** (Part 4 of 15)

Display 16.6 **Methods in the Classes ArrayList<T> and Vector<T>**

`public Vector(Collection<? extends T> c)`

Creates a vector that contains all the elements of the collection `c` in the same order as they have in `c`. In other words, the elements have the same index in the vector created as they do in `c`. This is not quite a true copy constructor because it does not preserve capacity. The capacity of the created vector will be `c.size()`, not `c.capacity`.
The vector created is only a shallow copy of the collection argument. The vector created contains references to the elements in `c` (not references to clones of the elements in `c`).
Throws a `NullPointerException` if `c` is `null`.

`public Vector(int initialCapacity, int capacityIncrement)`

Constructs an empty vector with the specified initial capacity and capacity increment. When the vector needs to grow, it will add room for `capacityIncrement` more items.
Throws an `IllegalArgumentException` if `initialCapacity` is negative.
(`ArrayList<T>` does not have a corresponding constructor.)

(continued)

Methods in the Classes **ArrayList<T>** and **Vector<T>** (Part 5 of 15)

Display 16.6 **Methods in the Classes ArrayList<T> and Vector<T>**

ARRAYLIKE METHODS FOR BOTH `ArrayList<T>` AND `Vector<T>`

`public T set(int index, T newElement)`

Sets the element at the specified `index` to `newElement`. The element previously at that position is returned. If you draw an analogy to an array `a`, this is analogous to setting `a[index]` to the value `newElement`. The `index` must be a value greater than or equal to `0` and strictly less than the current size of the list.
Throws an `IndexOutOfBoundsException` if the `index` is not in this range.

`public T get(int index)`

Returns the element at the specified index. This is analogous to returning `a[index]` for an array `a`. The `index` must be a value greater than or equal to `0` and less than the current size of the calling object.
Throws an `IndexOutOfBoundsException` if the `index` is not in this range.

(continued)

Methods in the Classes **ArrayList<T>** and **Vector<T>** (Part 6 of 15)

Display 16.6 **Methods in the Classes ArrayList<T> and Vector<T>**

METHODS TO ADD ELEMENTS FOR BOTH ArrayList<T> AND Vector<T>

```
public boolean add(T newElement)
```

Adds newElement to the end of the calling object's list and increases its size by 1. The capacity of the calling object is increased if that is required. Returns true if the add was successful. This method is often used as if it were a void method.

```
public void add(int index, T newElement)
```

Inserts newElement as an element in the calling object at the specified index and increases the size of the calling object by one. Each element in the calling object with an index greater than or equal to index is shifted upward to have an index that is one greater than the value it had previously.
The index must be a value greater than or equal to 0 and less than *or equal to* the size of the calling object (before this addition).
Throws an IndexOutOfBoundsException if the index is not in the prescribed range.
Note that you can use this method to add an element after the last current element. The capacity of the calling object is increased if that is required.

(continued)

Methods in the Classes **ArrayList<T>** and **Vector<T>** (Part 7 of 15)

Display 16.6 **Methods in the Classes ArrayList<T> and Vector<T>**

```
public boolean addAll(Collection<? extends T> c)
```

Appends all the elements in c to the end of the elements in the calling object in the order that they are enumerated by a c iterator. The behavior of this method is not guaranteed if the collection c is the calling object or any collection including the calling object either directly or indirectly.
Throws an NullPointerException if c is null.

```
public boolean addAll(int index, Collection<? extends T> c)
```

Inserts all the elements in c into the calling object starting at position index. Elements are inserted in the order that they are enumerated by a c iterator. Elements previously at positions index or higher are shifted to higher numbered positions.
Throws an IndexOutOfBoundsException if index is not both greater than or equal to zero and less than size().
Throws an NullPointerException if c is null.

(continued)

Methods in the Classes **ArrayList<T>** and **Vector<T>** (Part 8 of 15)

Display 16.6 **Methods in the Classes ArrayList<T> and Vector<T>**

METHODS TO REMOVE ELEMENTS FOR BOTH ArrayList<T> AND Vector<T>

```
public T remove(int index)
```

Deletes the element at the specified index and returns the element deleted. The size of the calling object is decreased by 1. The capacity of the calling object is not changed. Each element in the calling object with an index greater than or equal to index is decreased to have an index that is 1 less than the value it had previously.
The index must be a value greater than or equal to 0 and less than the size of the calling object (before this removal).
Throws an IndexOutOfBoundsException if the index is not in this range.

(continued)

Methods in the Classes **ArrayList<T>** and **Vector<T>** (Part 9 of 15)

Display 16.6 **Methods in the Classes ArrayList<T> and Vector<T>**

```
public boolean remove(Object theElement)
```

Removes the first occurrence of theElement from the calling object. If theElement is found in the calling object, then each element in the calling object with an index greater than or equal to theElement's index is decreased to have an index that is one less than the value it had previously. Returns true if theElement was found (and removed). Returns false if theElement was not found in the calling object. If the element was removed, the size is decreased by one. The capacity is not changed.

```
protected void removeRange(int fromIndex, int toIndex)
```

Removes all elements with index greater than or equal to fromIndex and strictly less than toIndex. Be sure to note that this method is protected, not public.

```
public void clear()
```

Removes all elements from the calling object and sets its size to zero.

(continued)

Methods in the Classes **ArrayList<T>** and **Vector<T>** (Part 10 of 15)

Display 16.6 **Methods in the Classes ArrayList<T> and Vector<T>**

SEARCH METHODS FOR BOTH ArrayList<T> AND Vector<T>

```
public boolean isEmpty()
```

Returns `true` if the calling object is empty (that is, has size 0); otherwise returns `false`.

```
public boolean contains(Object target)
```

Returns `true` if `target` is an element of the calling object; otherwise returns `false`. Uses the method `equals` of the object `target` to test for equality.

```
public int indexOf(Object target)
```

Returns the index of the first element that is equal to `target`. Uses the method `equals` of the object `target` to test for equality. Returns `-1` if `target` is not found.

(continued)

Methods in the Classes **ArrayList<T>** and **Vector<T>** (Part 11 of 15)

Display 16.6 **Methods in the Classes ArrayList<T> and Vector<T>**

```
public int lastIndexOf(Object target)
```

Returns the index of the last element that is equal to `target`. Uses the method `equals` of the object `target` to test for equality. Returns `-1` if `target` is not found.

ITERATORS FOR BOTH ArrayList<T> AND Vector<T>

```
public Iterator<T> iterator()
```

Returns an iterator for the calling object. Iterators are discussed in Section 16.2.

```
public ListIterator<T> listIterator()
```

Returns a `ListIterator<T>` for the calling object. `ListIterator<T>` is discussed in Section 16.2.

(continued)

Methods in the Classes **ArrayList<T>** and **Vector<T>** (Part 12 of 15)

Display 16.6 **Methods in the Classes ArrayList<T> and Vector<T>**

```
ListIterator<T> listIterator(int index)
```

Returns a list iterator for the calling object starting at `index`. The first element to be returned by the iterator is the one at `index`. (Iterators are discussed in Section 16.2.)
Throws an `IndexOutOfBoundsException` if `index` does not satisfy:

```
0 <= index <= size()
```

CONVERTING TO AN ARRAY FOR BOTH ArrayList<T> AND Vector<T>

```
public Object[] toArray()
```

Returns an array containing all of the elements in the calling object. The elements of the array are indexed the same as in the calling object.

(continued)

Methods in the Classes **ArrayList<T>** and **Vector<T>** (Part 13 of 15)

Display 16.6 **Methods in the Classes ArrayList<T> and Vector<T>**

```
public <E> E[] toArray(E[] a)
```

Note that the type parameter E is not the same as T. So, E can be any reference type; it need not be the type T in `Collection<T>`. For example, E might be an ancestor type of T.
Returns an array containing all of the elements in the calling object. The elements of the array are indexed the same as in the calling object.
The argument `a` is used primarily to specify the type of the array returned. The exact details are as follows:
The type of the returned array is that of `a`. If the collection fits in the array `a`, then `a` is used to hold the elements of the returned array; otherwise a new array is created with the same type as `a`.
If `a` has more elements than the calling object, then the element in `a` immediately following the end of the elements copied from the calling object are set to `null`.
Throws an `ArrayStoreException` if the type of `a` is not an ancestor type of the type of every element in the calling object.
Throws a `NullPointerException` if `a` is `null`.

(continued)

Methods in the Classes `ArrayList<T>` and `Vector<T>` (Part 14 of 15)

Display 16.6 **Methods in the Classes `ArrayList<T>` and `Vector<T>`**

MEMORY MANAGEMENT FOR BOTH `ArrayList<T>` AND `Vector<T>`

```
public int size()
```
Returns the number of elements in the calling object.

```
public int capacity()
```
Returns the current capacity of the calling object.

```
public void ensureCapacity(int newCapacity)
```
Increases the capacity of the calling object to ensure that it can hold at least `newCapacity` elements. Using `ensureCapacity` can sometimes increase efficiency, but its use is not needed for any other reason.

```
public void trimToSize()
```
Trims the capacity of the calling object to be the calling object's current size. This is used to save storage.

(continued)

Methods in the Classes `ArrayList<T>` and `Vector<T>` (Part 15 of 15)

Display 16.6 **Methods in the Classes `ArrayList<T>` and `Vector<T>`**

MAKE A COPY FOR BOTH `ArrayList<T>` AND `Vector<T>`

```
public Object clone()
```
Returns a shallow copy of the calling object.

Differences Between `ArrayList<T>` and `Vector<T>`

- For most purposes, the **`ArrayList<T>`** and **`Vector<T>`** are equivalent
  - The **`Vector<T>`** class is older, and had to be retrofitted with extra method names to make it fit into the collection framework
  - The **`ArrayList<T>`** class is newer, and was created as part of the Java collection framework
  - The **`ArrayList<T>`** class is supposedly more efficient than the **`Vector<T>`** class also

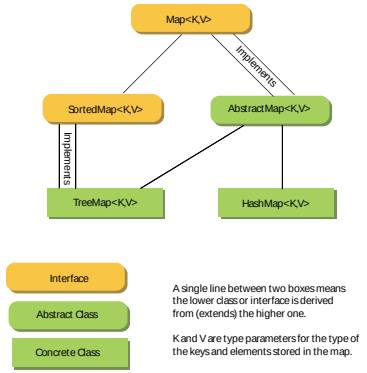
Pitfall: Omitting the <T>

- When the **`<T>`** or corresponding class name is omitted from a reference to a collection class, this is an error for which the compiler may or may not issue an error message (depending on the details of the code), and even if it does, the error message may be quite strange
  - Look for a missing **`<T>`** or **`<ClassName>`** when a program that uses collection classes gets a strange error message or doesn't run correctly

The Map Framework

- The Java *map* framework deals with collections of ordered pairs
  - For example, a key and an associated value
- Objects in the map framework can implement mathematical functions and relations, so can be used to construct database classes
- The map framework uses the **`Map<T>`** interface, the **`AbstractMap<T>`** class, and classes derived from the **`AbstractMap<T>`** class

The Map Landscape



The Map<K,V> Interface (1 of 3)

Display 16.9 Method Headings in the `Map<K,V>` Interface

The `Map<K,V>` interface is in the `java.util` package.

CONSTRUCTORS

Although not officially required by the interface, any class that implements the `Map<K,V>` interface should have at least two constructors: a no-argument constructor that creates an empty `Map<K,V>` object, and a constructor with one `Map<K,V>` parameter that creates a `Map<K,V>` object with the same elements as the constructor argument. The interface does not specify whether the copy produced by the one-argument constructor is a shallow copy or a deep copy of its argument.

METHODS

`boolean isEmpty( )`

Returns `true` if the calling object is empty; otherwise returns `false`.

`public boolean containsValue(Object value)`

Returns `true` if the calling object contains at least one or more keys that map to an instance of `value`.

`public boolean containsKey(Object key)`

Returns `true` if the calling object contains `key` as one of its keys.

The Map<K,V> Interface (2 of 3)

`public boolean equals(Object other)`

This is the `equals` of the map, not the `equals` of the elements in the map. Overrides the inherited method `equals`.

`public int size( )`

Returns the number of (key, value) mappings in the calling object.

`public int hashCode( )`

Returns the hash code value for the calling object.

`public Set<Map.Entry<K,V>> entrySet( )`

Returns a set *view* consisting of (key, value) mappings for all entries in the map. Changes to the map are reflected in the set and vice-versa.

`public Collection<V> values( )`

Returns a collection *view* consisting of all values in the map. Changes to the map are reflected in the collection and vice-versa.

`public V get(Object key)`

Returns the value to which the calling object maps `key`. If `key` is not in the map, then `null` is returned. Note that this does not always mean that the key is not in the map since it is possible to map a key to `null`. The `containsKey` method can be used to distinguish the two cases.

The Map<K,V> Interface (3 of 3)

OPTIONAL METHODS

The following methods are optional, which means they still must be implemented, but the implementation can simply throw an `UnsupportedOperationException` if, for some reason, you do not want to give the methods a "real" implementation. An `UnsupportedOperationException` is a `RunTimeException` and so is not required to be caught or declared in a `throws` clause.

```
public V put(K key, V value) (Optional)
```

Associates `key` to `value` in the map. If `key` was associated with an existing value then the old value is overwritten and returned. Otherwise `null` is returned.

```
public void putAll(Map<? extends K,? extends V> mapToAdd) (Optional)
```

Adds all mappings of `mapToAdd` into the calling object's map.

```
public V remove(Object key) (Optional)
```

Removes the mapping for the specified key. If the key is not found in the map then `null` is returned; otherwise the previous value for the key is returned.

Concrete Map Classes

- Normally you will use an instance of a Concrete Map Class
- Here we discuss the `HashMap<K, V>` Class
  - Internally, the class uses a hash table similar to what was discussed in Chapter 15.
  - No guarantee as to the order of elements placed in the map.
  - If you require order then you should use the `TreeMap<K,V>` class or the `LinkedHashMap<K,V>` class

HashMap<K,V> Class

- The initial capacity specifies how many "buckets" exist in the hash table.
  - This would be analogous to the size of the array of the hash table covered in Chapter 15.
  - A larger initial capacity results in faster performance but uses more memory
- The load factor is a number between 0 and 1.
  - This variable specifies a percentage such that if the number of elements added to the hash table exceeds the load factor then the capacity of the hash table is automatically increased.
- The default load factor is 0.75 and the default initial capacity is 16

The HashMap<K,V> Class (1 of 2)

Display 16.10 Methods in the `HashMap<K,V>` Class

The `HashMap<K,V>` class is in the `java.util` package.

The `HashMap<K,V>` class extends the `AbstractMap<K,V>` class and implements the `Map<K,V>` interface.

The `HashMap<K,V>` class implements all of the methods in the `Map<K,V>` interface (Display 16.9). The only other methods in the `HashMap<K,V>` class are the constructors.

All the exception classes mentioned are the kind that are not required to be caught in a `catch` block or declared in a `throws` clause.

All the exception classes mentioned are in the package `java.lang` and so do not require any import statement.

```
public HashMap( )
```

Creates a new, empty map with a default initial capacity of 16 and load factor of 0.75.

```
public HashMap(int initialCapacity)
```

Creates a new, empty map with a default capacity of `initialCapacity` and load factor of 0.75.

Throws a `IllegalArgumentException` if `initialCapacity` is negative.

```
public HashMap(int initialCapacity, float loadFactor)
```

Creates a new, empty map with the specified capacity and load factor.

Throws a `IllegalArgumentException` if `initialCapacity` is negative or `loadFactor` nonpositive.

The HashMap<K,V> Class (2 of 2)

`public HashMap(Map<? extends K,? extends V> m)`

Creates a new map with the same mappings as `m`. The `initialCapacity` is set to the same size as `m` and the `loadFactor` to 0.75.

Throws a `NullPointerException` if `m` is `null`.

`public Object clone( )`

Creates a shallow copy of this instance and returns it. The keys and values are not cloned.

The remainder of the methods are the same as those described for the `Map<K,V>` interface (Display 16.9.)

All of the Map Interface methods are supported, such as `get` and `put`

HashMap Example (1 of 3)

```
1       // This class uses the Employee class defined in Chapter 7.
2       import java.util.HashMap;
3       import java.util.Scanner;
4       public class HashMapDemo
5       {
6         public static void main(String[] args)
7         {
8               // First create a hashmap with an initial size of 10 and
9               // the default load factor
10              HashMap<String,Employee> employees =
11                      new HashMap<String,Employee>(10);

12              // Add several employees objects to the map using
13              // their name as the key
14              employees.put("Joe",
15                      new Employee("Joe",new Date("September", 15, 1970)));
16              employees.put("Andy",
17                      new Employee("Andy",new Date("August", 22, 1971)));
18              employees.put("Greg",
19                      new Employee("Greg",new Date("March", 9, 1972)));
20              employees.put("Kiki",
21                      new Employee("Kiki",new Date("October", 8, 1970)));
22              employees.put("Antoinette",
23                      new Employee("Antoinette",new Date("May", 2, 1959)));
24              System.out.print("Added Joe, Andy, Greg, Kiki, ");
25              System.out.println("and Antoinette to the map.");
```


HashMap Example (2 of 3)

```
26              // Ask the user to type a name.  If found in the map,
27              // print it out.
28              Scanner keyboard = new Scanner(System.in);
29              String name = "";
30              do
31              {
32                System.out.print("\nEnter a name to look up in the map. ");
33                System.out.println("Press enter to quit.");
34                name = keyboard.nextLine();
35                if (employees.containsKey(name))
36                {
37                  Employee e = employees.get(name);
38                  System.out.println("Name found: " + e.toString());
39                }
40                else if (!name.equals(""))
41                {
42                  System.out.println("Name not found.");
43                }
44              } while (!name.equals(""));
45      }
46 }
```


HashMap Example (3 of 3)

SAMPLE DIALOGUE

```
Added Joe, Andy, Greg, Kiki, and Antoinette to the map.

Enter a name to look up in the map. Press enter to quit.
Joe
Name found: Joe September 15, 1970

Enter a name to look up in the map. Press enter to quit.
Andy
Name found: Andy August 22, 1971

Enter a name to look up in the map. Press enter to quit.
Kiki
Name found: Kiki October 8, 1970

Enter a name to look up in the map. Press enter to quit.
Myla
Name not found.
```

Using HashMap with your own Class

- Just like the HashSet class, If you intend to use the HashMap<K,V> class with your own class as the parameterized type K , then your class must override the following methods:
  - public int hashCode();
    - Ideally returns a unique integer for this object
  - public boolean equals(Object obj);
    - Indicates whether or not the reference object is the same as the parameter obj

Iterators

- An iterator is an object that is used with a collection to provide sequential access to the collection elements
  - This access allows examination and possible modification of the elements
- An iterator imposes an ordering on the elements of a collection even if the collection itself does not impose any order on the elements it contains
  - If the collection does impose an ordering on its elements, then the iterator will use the same ordering

The `Iterator<T>` Interface

- Java provides an `Iterator<T>` interface
  - Any object of any class that satisfies the `Iterator<T>` interface is an `Iterator<T>`
- An `Iterator<T>` does not stand on its own
  - It must be associated with some collection object using the method `iterator`
  - If `c` is an instance of a collection class (e.g., `HashSet<String>`), the following obtains an iterator for `c`:

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


Methods in the `Iterator<T>` Interface (Part 1 of 2)

Methods in the `Iterator<T>` Interface

The `Iterator<T>` interface is in the `java.util` package.
All the exception classes mentioned are the kind that are not required to be caught in a `catch` block or declared in a `throws` clause.
`NoSuchElementException` is in the `java.util` package, which requires an import statement if your code mentions the `NoSuchElementException` class. All the other exception classes mentioned are in the package `java.lang` and so do not require any import statement.

```
public T next()
```

Returns the next element of the collection that produced the iterator.
Throws a `NoSuchElementException` if there is no next element.

(continued)

Methods in the `Iterator<T>` Interface (Part 2 of 2)

Methods in the `Iterator<T>` Interface

```
public boolean hasNext()
```

Returns `true` if `next()` has not yet returned all the elements in the collection; returns `false` otherwise.

```
public void remove() (Optional)
```

Removes from the collection the last element returned by `next`.
This method can be called only once per call to `next`. If the collection is changed in any way, other than by using `remove`, the behavior of the iterator is not specified (and thus should be considered unpredictable).
Throws `IllegalStateException` if the `next` method has not yet been called, or the `remove` method has already been called after the last call to the `next` method.
Throws an `UnsupportedOperationException` if the `remove` operation is not supported by this `Iterator<T>`.

Using an Iterator with a HashSet<T> Object

- A **`HashSet<T>`** object imposes no order on the elements it contains
- However, an iterator will impose an order on the elements in the hash set
  - That is, the order in which they are produced by **`next()`**
  - Although the order of the elements so produced may be duplicated for each program run, there is no requirement that this must be the case

An Iterator (Part 1 of 3)

An Iterator

```
1  import java.util.HashSet;
2  import java.util.Iterator;

3  public class HashSetIteratorDemo
4  {
5      public static void main(String[] args)
6      {
7          HashSet<String> s = new HashSet<String>();

8          s.add("health");
9          s.add("love");
10         s.add("money");

11         System.out.println("The set contains:");
```

(continued)

An Iterator (Part 2 of 3)

An Iterator

```
12         Iterator<String> i = s.iterator();
13         while (i.hasNext())
14             System.out.println(i.next());

15         i.remove();

16         System.out.println();
17         System.out.println("The set now contains:");

18         i = s.iterator();
19         while (i.hasNext())
20             System.out.println(i.next());

21         System.out.println("End of program.");
22     }
23 }
```

You cannot "reset" an iterator "to the beginning." To do a second iteration, you create another iterator.

(continued)

An Iterator (Part 3 of 3)

An Iterator

SAMPLE DIALOGUE

```
The set contains:
money
love
health

The set now contains:
money
love
End of program.
```

The HashSet<T> object does not order the elements it contains, but the iterator imposes an order on the elements.

Tip: For-Each Loops as Iterators

- Although it is not an iterator, a for-each loop can serve the same purpose as an iterator
  - A for-each loop can be used to cycle through each element in a collection
- For-each loops can be used with any of the collections discussed here

For-Each Loops as Iterators (Part 1 of 2)

For-Each Loops as Iterators

```
1  import java.util.HashSet;
2  import java.util.Iterator;

3  public class ForEachDemo
4  {
5      public static void main(String[] args)
6      {
7          HashSet<String> s = new HashSet<String>();

8          s.add("health");
9          s.add("love");
10         s.add("money");

11         System.out.println("The set contains:");
```

(continued)

For-Each Loops as Iterators (Part 2 of 2)

For-Each Loops as Iterators

```
12         String last = null;
13         for (String e : s)
14         {
15             last = e;
16             System.out.println(e);
17         }

18         s.remove(last);

19         System.out.println();
20         System.out.println("The set now contains:");

21         for (String e : s)
22             System.out.println(e);

23         System.out.println("End of program.");
24     }
25 }
```

The output is the same as in Display 16.8.

The `ListIterator<T>` Interface

- The **`ListIterator<T>`** interface extends the **`Iterator<T>`** interface, and is designed to work with collections that satisfy the **`List<T>`** interface
  - A **`ListIterator<T>`** has all the methods that an **`Iterator<T>`** has, plus additional methods
  - A **`ListIterator<T>`** can move in either direction along a list of elements
  - A **`ListIterator<T>`** has methods, such as **`set`** and **`add`**, that can be used to modify elements

Methods in the `ListIterator<T>` Interface (Part 1 of 4)

Methods in the `ListIterator<T>` Interface

The `ListIterator<T>` interface is in the `java.util` package.
The *cursor position* is explained in the text and in Display 16.11.
All the exception classes mentioned are the kind that are not required to be caught in a `catch` block or declared in a `throws` clause.
`NoSuchElementException` is in the `java.util` package, which requires an import statement if your code mentions the `NoSuchElementException` class. All the other exception classes mentioned are in the package `java.lang` and so do not require any import statement.

`public T next()`

Returns the next element of the list that produced the iterator. More specifically, returns the element immediately after the cursor position.
Throws a `NoSuchElementException` if there is no next element.

(continued)

Methods in the `ListIterator<T>` Interface (Part 2 of 4)

Methods in the `ListIterator<T>` Interface

`public T previous()`

Returns the previous element of the list that produced the iterator. More specifically, returns the element immediately before the cursor position.
Throws a `NoSuchElementException` if there is no previous element.

`public boolean hasNext()`

Returns `true` if there is a suitable element for `next()` to return; returns `false` otherwise.

`public boolean hasPrevious()`

Returns `true` if there is a suitable element for `previous()` to return; returns `false` otherwise.

`public int nextIndex()`

Returns the index of the element that would be returned by a call to `next()`. Returns the list size if the cursor position is at the end of the list.

(continued)

Methods in the `ListIterator<T>` Interface (Part 3 of 4)

Methods in the `ListIterator<T>` Interface

`public int previousIndex()`

Returns the index that would be returned by a call to `previous()`. Returns −1 if the cursor position is at the beginning of the list.

`public void add(T newElement)` *(Optional)*

Inserts `newElement` at the location of the iterator cursor (that is, before the value, if any, that would be returned by `next()` and after the value, if any, that would be returned by `previous()`).
Cannot be used if there has been a call to `add` or `remove` since the last call to `next()` or `previous()`.
Throws `IllegalStateException` if neither `next()` nor `previous()` has been called, or the `add` or `remove` method has already been called after the last call to `next()` or `previous()`.
Throws an `UnsupportedOperationException` if the `remove` operation is not supported by this `Iterator<T>`.
Throws a `ClassCastException` if the class of `newElement` prevents it from being added.
Throws an `IllegalArgumentException` if some property other than the class of `newElement` prevents it from being added.

(continued)

Methods in the `ListIterator<T>` Interface (Part 4 of 4)

Methods in the `ListIterator<T>` Interface

`public void remove()` *(Optional)*

Removes from the collection the last element returned by `next()` or `previous()`.
This method can be called only once per call to `next()` or `previous()`.
Cannot be used if there has been a call to `add` or `remove` since the last call to `next()` or `previous()`.
Throws `IllegalStateException` if neither `next()` nor `previous()` has been called, or the `add` or `remove` method has already been called after the last call to `next()` or `previous()`.
Throws an `UnsupportedOperationException` if the `remove` operation is not supported by this `Iterator<T>`.

`public void set(T newElement)` *(Optional)*

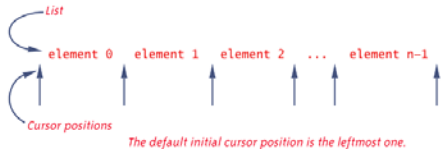
Replaces the last element returned by `next()` or `previous()` with `newElement`.
Cannot be used if there has been a call to `add` or `remove` since the last call to `next()` or `previous()`.
Throws an `UnsupportedOperationException` if the `set` operation is not supported by this `Iterator<T>`.
Throws `IllegalStateException` if neither `next()` nor `previous()` has been called, or the `add` or `remove` method has been called since the last call to `next()` or `previous()`.
Throws an `ClassCastException` if the class of `newElement` prevents it from being added.
Throws an `IllegalArgumentException` if some property other than the class of `newElement` prevents it from being added.

The `ListIterator<T>` Cursor

- Every **`ListIterator<T>`** has a position marker known as the *cursor*
  - If the list has *n* elements, they are numbered by indices 0 through *n*-1, but there are *n*+1 cursor positions
  - When **`next()`** is invoked, the element immediately following the cursor position is returned and the cursor is moved forward one cursor position
  - When **`previous()`** is invoked, the element immediately before the cursor position is returned and the cursor is moved back one cursor position

`ListIterator<T>` Cursor Positions

`ListIterator<T>` Cursor Positions



Pitfall: `next` and `previous` Can Return a Reference

- Theoretically, when an iterator operation returns an element of the collection, it might return a copy or clone of the element, or it might return a reference to the element
- Iterators for the standard predefined collection classes, such as **`ArrayList<T>`** and **`HashSet<T>`**, actually return references
  - Therefore, modifying the returned value will modify the element in the collection

An Iterator Returns a Reference (Part 1 of 4)

An Iterator Returns a Reference

```
1  import java.util.ArrayList;
2  import java.util.Iterator;
3  public class IteratorReferenceDemo
4  {
5      public static void main(String[] args)
6      {
7          ArrayList<Date> birthdays = new ArrayList<Date>();

8          birthdays.add(new Date(1, 1, 1990));
9          birthdays.add(new Date(2, 2, 1990));
10         birthdays.add(new Date(3, 3, 1990));

11         System.out.println("The list contains:");
```

The class `Date` *is defined in Display 4.13, but you can easily guess all you need to know about* `Date` *for this example.*

(continued)

An Iterator Returns a Reference (Part 2 of 4)

An Iterator Returns a Reference

```
12         Iterator<Date> i = birthdays.iterator();
13         while (i.hasNext())
14             System.out.println(i.next());

15         i = birthdays.iterator();
16         Date d = null; //To keep the compiler happy.
17         System.out.println("Changing the references.");
18         while (i.hasNext())
19         {
20             d = i.next();
21             d.setDate(4, 1, 1990);
22         }
```

(continued)

An Iterator Returns a Reference (Part 3 of 4)

An Iterator Returns a Reference

```
23         System.out.println("The list now contains:");

24         i = birthdays.iterator();
25         while (i.hasNext())
26             System.out.println(i.next());

27         System.out.println("April fool!");
28     }
29 }
```

(continued)

An Iterator Returns a Reference (Part 4 of 4)

An Iterator Returns a Reference

SAMPLE DIALOGUE

```
The list contains:
January 1, 1990
February 2, 1990
March 3, 1990
Changing the references.
The list now contains:
April 1, 1990
April 1, 1990
April 1, 1990
April fool!
```

Tip: Defining Your Own Iterator Classes

- There is usually little need for a programmer defined `Iterator<T>` or `ListIterator<T>` class
- The easiest and most common way to define a collection class is to make it a derived class of one of the library collection classes
  - By doing this, the `iterator()` and `listIterator()` methods automatically become available to the program
- If a collection class must be defined in some other way, then an iterator class should be defined as an inner class of the collection class