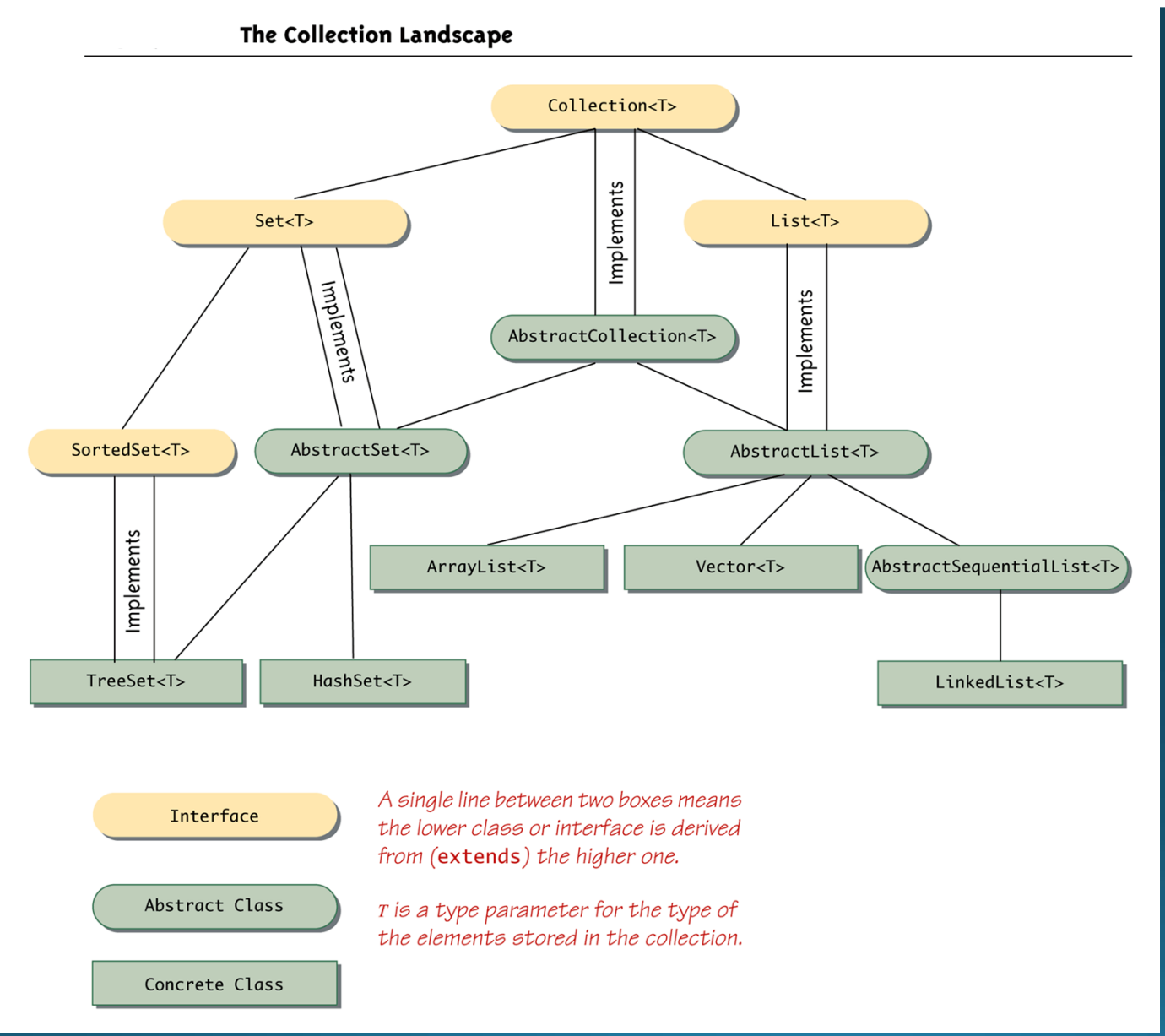
Java Collections

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



Adapted from material by Walter Savitch

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)

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)

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)

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)

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.

```
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)

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)

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)

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)

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)

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)

Adapted from material by Walter Savitch

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

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. 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.

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)

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)

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)

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)

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)

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)

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)

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)

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)

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)

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)

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)

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)

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)

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)

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)

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)

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

HashSet<T> Class Demo (1 of 4)

```
1  import java.util.HashSet;
2  import java.util.Iterator;
3  public class HashSetDemo {
4      private static void outputSet(HashSet<String> set) {
5          Iterator<String> i = set.iterator( );
6          while (i.hasNext( ))
7              System.out.print(i.next( ) + " ");
8          System.out.println();
9      }
10
11     public static void main(String[] args) {
12         HashSet<String> round = new HashSet<String>( );
13         HashSet<String> green = new HashSet<String>( );
14
15         // Add some data to each set
16         round.add("peas");
17         round.add("ball");
18         round.add("pie");
19         round.add("grapes");
20
```

Adapted from material by Walter Savitch

HashSet<T> Class Demo (2 of 4)

```
21          System.out.println("Contents of set round: ");
22          outputSet(round);
23          System.out.println("\nContents of set green: ");
24          outputSet(green);
25
26          System.out.println("\nball in set 'round'? " + round.contains("ball"));
27          System.out.println("ball in set 'green'? " + green.contains("ball"));
28
29          System.out.println("\nball and peas in same set? " +
30              ((round.contains("ball") && (round.contains("peas"))) ||
31                  (green.contains("ball") && (green.contains("peas")))));
32          System.out.println("pie and grass in same set? " +
33              ((round.contains("pie") && (round.contains("grass"))) ||
34                  (green.contains("pie") && (green.contains("grass")))));
35
```

HashSet<T> Class Demo (3 of 4)

```
36          // To union two sets we use the addAll method.
37          HashSet<String> setUnion = new HashSet<String>(round);
38          round.addAll(green);
39          System.out.println("\nUnion of green and round:");
40          outputSet(setUnion);
41
42          // To intersect two sets we use the removeAll method.
43          HashSet<String> setInter = new HashSet<String>(round);
44          setInter.removeAll(green);
45          System.out.println("\nIntersection of green and round:");
46          outputSet(setInter);
47          System.out.println();
48      }
49  }
```

HashSet<T> Class Demo (4 of 4)

SAMPLE OUTPUT

```
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

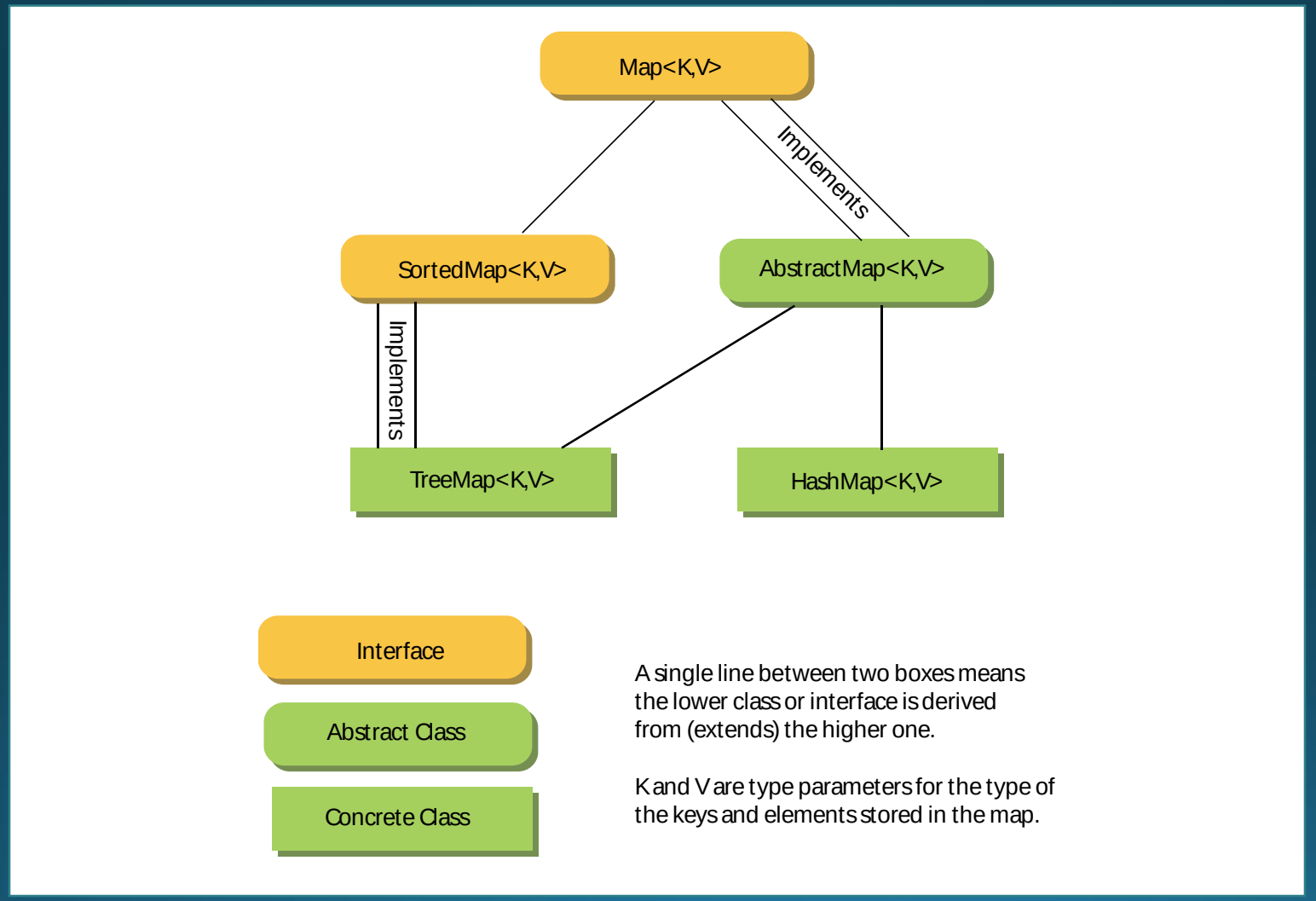
Pitfall: Omitting the <T>

- 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



Adapted from material by Walter Savitch

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

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.

Adapted from material by Walter Savitch

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.

Adapted from material by Walter Savitch

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.

Adapted from material by Walter Savitch

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
  - 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
    - LinkedHashMap: Hash table and linked list implementation of the Map interface, with predictable iteration order. This implementation differs from HashMap in that it maintains a doubly-linked list running through all of its entries.
    - TreeMap: The map is sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time, depending on which constructor is used. This implementation provides guaranteed log(n) time cost for the containsKey, get, put and remove operations.

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)

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.

Adapted from material by Walter Savitch

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

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      public static void main(String[] args){
6          // First create a hashmap with an initial size of 10 and
7          // the default load factor
8          HashMap<String,Employee> employees = new HashMap<String,Employee>(10);
9
10         // Add several employees objects to the map using
11         // their name as the key
12         employees.put("Joe", new Employee("Joe",new Date("September", 15, 1970)));
13         employees.put("Andy", new Employee("Andy",new Date("August", 22, 1971)));
14         employees.put("Greg", new Employee("Greg",new Date("March", 9, 1972)));
15         employees.put("Kiki", new Employee("Kiki",new Date("October", 8, 1970)));
16         employees.put("Antoinette", new Employee("Antoinette",new Date("May", 2, 1959)));
17         System.out.print("Added Joe, Andy, Greg, Kiki, ");
18         System.out.println("and Antoinette to the map.");
19
```

HashMap Example (2 of 3)

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

HashMap Example (3 of 3)

SAMPLE OUTPUT

```
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