COMPUTING INTEGRITY
INCORPORATED
60 Belvedere Avenue
Point Richmond, CA 94801-4023
510.233.5400 Sales
510-233.5444 Support
510.233.5446 Facsimile

PROGRESS SOFTWARE BUSINESS ALLIANCE
ISV Partner

Collection and Map Classes for OOABL – Version 0.1

27 April 2006

Revision History

Version 0.1 – 27 April 2006 – Initial release.

Background

In developing a foundation framework for Object-Oriented ABL, it seems natural to consider creating a set of Collection classes since they have a broad utility in OO design in other languages. It also seems natural to consider imitating the Collection classes in Java[1] since that is a tried and true implementation into which there has been a considerable investment of thought and effort. However, while Java Collection classes need to use relatively low level constructs like arrays and hash tables for implementation, in OOABL we have temp-tables, which not only provide a very easy and flexible way of maintaining a set, but which has the significant advantage of being open-ended (unlike an array) and having an automatic extension to disk. This use of temp-tables is likely to lead to a number of appropriate differences between a good OOABL implementation and the Java implementation.

Since Java Collection classes are a class hierarchy in which there are progressively more assumptions about the set, it seems likely that the use of temp-tables will obviate the need for some of the intermediate classes since there is no advantage to a defining a degenerate version once one is already using full featured temp-tables. I.e., if one has a concrete implementation using temp-tables, providing an alternate concrete implementation that also used temp-tables, but included less capability would be pointless. An example of this kind of distinction is that some Java Collection classes only provide for forward navigation in the Collection, but all of the OOABL classes described here provide both forward and backward navigation because it is trivial to include (i.e., Iterator versus ListIterator).

Another difference will be required due to the lack of support for method overloading in the current OOABL, as some of the standard Java methods for Collection classes exploit overloading[2]. E.g., add(Object elem) as a method defined in the Collection interface becomes add(int index, Object elem) in the List implementation. This will require either eliminating one form or adopting a different naming structure for methods in current versions of ABL. Likewise, some of the names used in the Java Collection classes for either the classes or the methods are reserved words for ABL so different names will be required or those classes or methods will need to be skipped.

Furthermore, some of the Java concrete implementations are distinguished by the method of implementation, e.g., HashSet or TreeSet. Since we will use temp-tables throughout, these variations are meaningless and those classes will be omitted in favor of a single concrete

[1] My reference for the Java implementation is The Java Programming Language: Third Edition by Ken Arnold, James Gosling, and David Holmes.
[2] In Java 1.5 the use of generics has significantly simplified the Java Collection class implementation, but there is no counterpart in OOABL at this time either.

implementation per type using temp-tables. This will result in some shift in what is an interface and what is a concrete class relative to the Java example.

Also, Java Collections and Maps are collections of Objects, but Java includes a full set of object versions of primitive data types, e.g,. Integer versus integer. Consequently, Java Collections are suitable for collection of lists of strings and the like and the key portion of a Map can consistently be an object, even if it is a name or sequence number. In OOABL we do not have these object versions of primitive types and, even if we were to create them, they would not sort in the expected way in the context of a temp-table. To get desired sorting we would have to revert to low level comparators and the like instead of using temp-tables It seems that, in the context of OOABL, a lightweight Collection such as a list of names is probably handled with a correspondingly lightweight implementation such as a local temp-table or delimited list. Thus, Collections will be currently used only for sets of true objects. In the case of Maps, the necessity of having keys of the appropriate type will require us to create a number of concrete classes where Java has only one.

With all of these qualifications, one might begin to wonder if the decision to use temp-tables was an appropriate one since it appears to be forcing a fair number of differences from the Java base that has been proposed as a model. While it is undoubtedly the case that the low level implementation provided by Java has some advantages, on the whole we feel that the added functionality provided by temp-tables is compelling. While it is good to seek inspiration and guidance from best practice in other OO languages, it is important to recognize that the spirit of a 4GL is different from a 3GL and that one should expect some differences in practice because of this. An excellent example of this is the ProDataSet, which in Java would involve considerable code, but whose basic functionality is available to use in ABL as a "primitive" for which coding is only required for overriding basic functionality. Thus, the goal in the current work is to strike a balance in providing a generic set of foundation classes that can be simply used in the general case to achieve reasonably high levels of functionality without additional coding. This does not mean that there will not be some cases in which special coding is required.

Exceptions

One of the obstacles in OO programming in ABL is the lack of an exception mechanism. Nothing that one can do in the language can raise an error condition in the same way that a Java exception does[3]. CI has created a non-OO exception handling mechanism that requires the calling program to check for errors, but otherwise has a number of desirable features such as the ability to record both fatal and non-fatal errors, to stack multiple errors, and to propagate errors back up the call stack. Thus, any given level in the call stack can handle any errors that are within its competency, or it can propagate the error to a higher level without having to reapply the error. We will be implementing an OO version of this error handling approach and will retrofit it to these Collection classes when it becomes available. Earlier versions of the Collection classes will contain comments or references in this document indicating where exception calls would occur. In the initial release, this distinction is not notable because the general response to an unexpected condition is simply to make a null return. It is expected to enhance this response with subsequent releases when the error handler is available.

[3] This is not to suggest that we find the form of the Java exception mechanism to be especially desirable, but it is better to have something than nothing.

Iteration

In the standard Java implementation, the Collection interface defines an `Iterator()` method that returns an Iterator class, which can be used for navigating the Collection. The basic Iterator class provides only forward movement through a Collection while a ListIterator provides both forward and backward movement. Because we are electing to use temp-tables for the concrete implementation, we have decided to incorporate the Iterator methods into the Collection interface and to provide ListIterator functionality for all concrete classes. We considered separating the Collection from the Iterator, but this would have required a two stage lookup and no particular advantages. The apparent potential advantage of the separation would be to have two Iterators simultaneously for the same Collection, but it isn't clear how one can keep the Iterators in sync with the Collection if this were the case. Thus, there being no apparent advantage, we elected to fold the functionality into the Collection classes themselves.

Compare() and compareTo()

These methods depend entirely on the specific domain. I.e., they would not be implemented in the base Collection classes, but in domain-specific classes derived from them. Java provides a java.lang.Comparable interface containing a single method equivalent to the OOABL:

```
method public integer CompareTo( input mob_Other as class Progress.Lang.Object ).
```

This method is implemented by any given domain class to return a -1, 0, or 1 indicating whether the current object is less than, equal to, or greater than the mob_Other object according to the "natural" order of these objects. When a domain class does not implement Comparable or the "natural" order is incorrect for some purpose, one provides a class which implements java.util.Comparator instead. This interface also contains a single method:

```
method public integer Compare(
    input mob_Object1 as class Progress.Lang.Object
    input mob_Object2 as class Progress.Lang.Object ).
```

This method returns the same values as CompareTo() where mob_Object1 takes the place of the current object in CompareTo(). Many domains will require multiple Comparator objects because there are many possible orderings, i.e., multiple CompareByX() methods.

In the current release we have elected to ignore the question of Compare() nd CompareTo() on the grounds that sorting on any desired key is easily accomplished by selecting a Map class of the appropriate type and assigning the desired keys. I.e., instead of the pair-wise comparison required in Java, the temp-table provides this sorting through a sophisticated, built-in indexing mechanism, making the Java comparison structure unnecessary.

Note, however, that if OOABL were to be extended so that there were object equivalents of primitives, e.g., Integer for integer, it would be tempting to simplify the Map classes described below into two classes which used objects for keys. However, this would only be possible if temp-table functionality were extended to include sorting these objects according to their value instead of according to their handle as is now the case.

Use of Progress.Lang.Object

In the current version of OOABL (10.1A), one cannot create temp-tables containing a specific object type. Instead, one can only create temp-tables that contain the base class Progress.Lang.Object, which is the superclass of all ABL objects. Therefore, there are no issues of type conflict in the

Collection classes since there is only this base class available. Thus, it becomes the responsibility of the class using the Collection to maintain consistent types. While this might seem to be a limitation, it also is what allows a generic Collection class instead of separate Collection class per object type. Collection classes in OOABL are quite capable of being of mixed type, but the programmer would have to assure the correct `CAST()` in use.

We have considered the idea of passing a class name to the Collection class for the purpose of making a cast, but, since it is not possible to dynamic change the return type of a method, already "pre-`cast()`" objects could not be returned without creating a domain-specific Collection class. Therefore, for the present, we are electing to make these Collection classes entirely generic and to place the responsibility for `cast()`ing objects to the correct domain class on the user of the Collection class. Similarly, it is the user of the class that will determine properties of the key for a Map so that this can remain generic.

Collection Versus Map

The Java Collection Class hierarchy actually has two separate class hierarchies, one derived from the Collection interface and one derived from the Map interface. The distinction between these is that a Collection is created by adding individual objects, but a Map is created by adding key/value pairs where the key is used to identify the object.

The Collection Interface

The foundation of one hierarchy of the classes in this group is the Collection interface. This interface defines the following methods:

```
method public integer size().
method public logical isEmpty().
method public logical containsElement(
    input mob_Element as class Progress.Lang.Object ).
method public logical addElement(
    input mob_Element as class Progress.Lang.Object ).
method public logical removeElement(
    input mob_Element as class Progress.Lang.Object ).
/*
method public logical containsAll(
    input mob_Collection as class com.cintegrity.Libl.Collection.Collection ).
method public logical addAll(
    input mob_Collection as class com.cintegrity.Libl.Collection.Collection ).
method public logical removeAll(
    input mob_Collection as class com.cintegrity.Libl.Collection.Collection ).
method public logical retainAll(
    input mob_Collection as class com.cintegrity.Libl.Collection.Collection ).
*/
method public void clearAll().
method public logical hasNextElement().
method public class Progress.Lang.Object nextElement( ).
method public void hasPrevElement( ).
method public class Progress.Lang.Object prevElement( ).
method public class Progress.Lang.Object setElement(
    input mob_Element as class Progress.Lang.Object ).
```

Because of a number of reserved word conflicts, the word "Element" has been consistently added to many method names when compared to the parallel Java method. The last five methods derive from Iterator rather than Collection in the Java construct and consequently the `Iterator()` method is not included in the interface. The implementation of the methods shown in red is being deferred

to a second phase in order to gain some early experience before attempting them. The method `toArray()` is not implemented since arrays are less desirable than temp-tables in virtually all cases.

In general, the functionality of these methods corresponds to the Java method with the corresponding name. Briefly, this is:

`size()`	Returns the number of elements in the Collection.
`isEmpty()`	Returns true if the Collection has no Elements.
`containsElement()`	Returns true if the Element is found in the Collection.
`addElement()`	Adds the Element to the Collection at the end; returns False if the Element was already in the Collection, otherwise True.
`removeElement()`	Removes the current Element from the Collection; returns False if there was no current Element.
`containsAll()`	Returns True if all Elements in the parameter are already in the Collection.
`addAll()`	Adds all Elements in the parameter to the Collection; returns True if any new Elements were added.
`removeAll()`	Removes all Elements in the parameter from the Collection; returns True is any Elements were removed.
`retainAll()`	Removes any Element in the Collection which are not in the parameter; returns True if any Elements were removed.
`clearAll()`	Removes all Elements from the Collection.
`hasNextElement()`	Returns True if there is another Element beyond the current one.
`nextElement()`	Returns the next Element beyond the current one; raises "No Such Element" exception and returns null if not available.
`hasPrevElement()`	Returns True if there is another Element prior to the current one.
`prevElement()`	Returns the prior Element to the current one; raises "No Such Element" exception and returns null if not available.
`setElement()`	Replaces the current Element and returns the Element replaced; raises "No Such Element" exception and returns null if not available.

Set and SortedSet

In the Java Collection hierarchy, Set is an interface that extends Collection to specify that there are no duplicates and SortedSet extends Set to specify that Iterators will return the members of the set in a specified order. Since "set" is an ABL reserved word that cannot be used for a class or interface name and any Set implemented with temp-table will necessarily return objects in a predictable order, we are electing not to implement Set as a separate class or interface, but will only implement SortedSet. SortedSet will be implemented as a class, not an interface, because there are no alternate concrete implementations derive from it (see below).

SortedSet provides the following additional methods:

```
method public class Progress.Lang.Object getFirst().
method public class Progress.Lang.Object getLast().
/*
method public class com.cintegrity.util.SortedSet subSet(
    input mob_MinObject as class Progress.Lang.Object
    input mob_MaxObject as class Progress.Lang.Object ).
method public class com.cintegrity.util.SortedSet headSet(
    input mob_MaxObject as class Progress.Lang.Object ).
method public class com.cintegrity.util.SortedSet tailSet(
    input mob_MinObject as class Progress.Lang.Object ).
method public class com.cintegrity.util.SortedSet copySubSet(
```

```
    input mob_MinObject as class Progress.Lang.Object
    input mob_MaxObject as class Progress.Lang.Object ).
method public class com.cintegrity.util.SortedSet copyHeadSet(
    input mob_MaxObject as class Progress.Lang.Object ).
method public class com.cintegrity.util.SortedSet copyTailSet(
    input mob_MinObject as class Progress.Lang.Object ).
*/
```

The prefix "get" has been added to the Java `first()` and `last()` methods because these are reserved words. The classes `subSet()`, `headSet()`, and `tailSet()` are implemented in Java as a view *backed by* the full set, i.e., changes in them will be reflected in the main set. This apparently will require a class SubSortedSet that will have a connection to the original SortedSet in order to keep the synchronization. The `copyXxxxSet()` methods are normally a part of the classes derived from Set and SortedSet, but are included here since those classes will not be implemented. The implementation of the methods shown in red is being deferred to a second phase in order to gain some early experience before attempting them. The Java method `comparator()` is not implemented because of the assumption that any ordering is provided by the user of this class.

In general, the functionality of these methods corresponds to the Java method with the corresponding name. Briefly, this is:

`getFirst()`	Returns the first Element in the Collection.
`getLast()`	Returns the last Element in the Collection.
`subSet()`	Returns a view onto the Collection consisting only of the Elements bounded by the two parameters; returns an Illegal Argument Exception if the range of the parameters exceeds the range of the Collection.
`headSet()`	Returns a view onto the Collection consisting only of the Elements from the lowest up to the value specified by the parameter; returns an Illegal Argument Exception if the range of the parameter exceeds the range of the Collection.
`tailSet()`	Returns a view onto the Collection consisting only of the Elements from the lowest up to the value specified by the parameter; returns an Illegal Argument Exception if the range of the parameter exceeds the range of the Collection.
`copySubSet()`	Like subSet(), only it returns an independent copy, not a view.
`copyHeadSet()`	Like headSet(), only it returns an independent copy, not a view.
`copyTailSet()`	Like tailSet(), only it returns an independent copy, not a view.

HashSet and TreeSet

In Java HashSet is a concrete implementation of Set using hashtables and TreeSet is a concrete implementation of SortedSet that uses a balanced tree. Since we are using temp-tables for the implementation throughout, there is no need for either concrete class, so the concrete implementation will be provided by SortedSet.

List

In Java, List is an interface that extends Collection to define a set that has a defined order and position, i.e., each object can be considered the Nth object in the set. For Java, Nth goes from 0 to `List:size()` – 1, but in keeping with Progress standards we will use 1 through `List:size()`. Additions to the List occur at the end and deletion of an object causes all higher objects to shift down by one. Note that List does not have the condition that List Elements must be unique. The

association of an Element with a index position raises naming issues since many methods which List implements are overloaded versions of methods in the Collection interface, e.g., `add(Object)` in the Collection interface becomes `add(Index, Object)` in the List class. Without overloading, this requires a name change so we have consistently added "Nth" to the names used in the Collection interface. Naturally, this implies that there are two versions, one without the assumption of position as provided by the interface, and one with a position as specific to List. For consistency, we have included the word Element as used in the Collection interface.

```
method public class Progress.Lang.Object getNthElement(
    input min_Index as integer).
method public class Progress.Lang.Object setNthElement(
    input min_Index as integer,
    input mob_Object as class Progress.Lang.Object).
method public void addNthElement(
    input min_Index as integer,
    input mob_Object as class Progress.Lang.Object).
method public class Progress.Lang.Object removeNthElement(
    input min_Index as integer ).
method public integer indexOf(
    input mob_Object as class Progress.Lang.Object).
method public index lastIndexOf(
    input mob_Object as class Progress.Lang.Object).
/*
method public class com.cintegrity.util.List subList(
    input min_MinIndex as integer.
    input min_MaxIndex as integer).
*/
```

The class `subList()` is implemented in Java as a view *backed by* the full set, i.e., changes in it will be reflected in the main set. This apparently will require a class SubList that will have a connection to the original List in order to keep the synchronization. Implementation of `subList()` is deferred. The `listInterator()` method is not included because we do not intend to implement separate Iterator classes.

`getNthElement()`	Returns the Nth Element in the List; raises No Such Element exception if not present.
`setNthElement()`	Replaces the Nth Element in the List; returns replaced object.
`addNthElement()`	Adds Element at the Nth position, pushing up all above that.
`removeNthElement()`	Removes the Nth Element, shifting all above that down one.
`indexOf()`	Returns the lowest position of the Element in the List; raises No Such Element exception if it does not exist.
`lastIndexOf(}`	Returns the highest position of the Element in the List; raises No Such Element exception if it does not exist.
`subList()`	Returns a List containing the range of Elements specified by the parameters; raises No Such Element exception the range is not valid.

ArrayList and LinkedList

In Java, ArrayList is a concrete implementation of List using an array and LinkedList is a concrete implementation of List using a linked list. As we are using temp-tables, neither will be implemented and we have made List a concrete class instead of an interface.

Map and Sorted Map

Map is not an extension of Collection because one does not add isolated Elements to it, as one does with a Collection, but rather adds key/value pairs, thus allowing one to find the value using the key. Any given key Maps to only one value, but any given value may be Mapped to by more than one key. SortedMap extends Map to specify that the Map is sorted. Because we are using temp-tables and the key will be indexed for retrieval, there is no benefit to implementing these as separate interfaces or classes. One could implement Map with work-table, but work-table is inferior or equal to temp-table in all respects so there is no point in having them separate.

For the ABL implementation of Map, we are electing to introduce two changes to the packaging used in the Java example. The first of these relates to the restriction in Java that each Key be unique. While this is certainly appropriate for many instances, it is not appropriate for a case like a Collection of Customers with a Key of Name because we cannot guarantee that Name will be unique. Therefore, we will define two hierarchies under Map, one called SetMap and the other called ListMap based on the parallel to Set and List. SetMap will restrict Keys to be unique and ListMap will not.

The second change is based on the desire to make effective use of temp-tables for holding the data. To order data on any type of key, there are a limited number of choices if we want to utilize the built-in indexing of temp-tables in order to provide the sort. Since it seems desirable at a minimum to support both an object key and a character key, we are electing to provide separate class implementations for object, character, integer, decimal, date, and datetime rather than relying on some artificial mechanism like forcing the user to format non-character data into strings appropriate for sorting.

Map is a reserved word and cannot be used as the name of a class or interface, so our original inclination was to define the methods one would expect to find defined in Map in SetMap and ListMap. However, most of the methods that one would like to include in these two interfaces require either the specification of a method return type or a method parameter corresponding to the type of the key. This means that these methods can be defined no higher than a branch in which the type of the key is defined, which in our case is the concrete class. This means that there is only one method difference between ListMap and SetMap in the methods that can be defined in an interface shared by all members of either branch.

Moreover, the strong parallels between all Map classes, differing primarily in the type of the key and the small difference of ListMap versus SetMap and the lack of a way to express this communality meaningfully in an inheritance or interface hierarchy has driven us to utilize a programming approach which we normally eschew, i.e., the use of an include file with preprocessor directives to hold the superset of logic for all Map classes. While this simplifies maintenance and facilitates extending Map classes to additional key datatypes, it does negatively impact the readability of the code. We can only suggest that anyone wishing to study the code of an individual class in a more readable fashion should compile with the preprocess option in order to see a "clean" copy of the code for that class.

Thus, the structure of the Map hierarchy is a single interface, BaseMap and one concrete class for each implemented key datatype and ListMap versus SetMap option. All of the concrete classes consist of a single include file reference to ObjectBaseMap.i. Object BaseMap.i is so named because the value portion of the Key/Value pair is defined as an object, as discussed above. Were one to

decide to implement other datatype values, one could adapt this include file accordingly. The naming of all concrete Map classes is [key datatype]Object[List|Set]Map, e.g., StringObjectListMap or IntegerObjectSetMap. Not all datatype variations have been implemented because they seem to be of questionable value, but they could very quickly be added to this structure if needed.

The methods of the BaseMap interface are:

```
method public integer size().
method public logical isEmpty().
method public logical containsValue(
    input mob_Value as class Progress.Lang.Object ).
method public void clearAll().
/*
method public class com.cintegrity.util.SortedSet keySet().
method public class com.cintegrity.util.Collection values().
method public class com.cintegrity.util.SortedSet entrySet().
*/
method public logical hasNextKey().
method public class Progress.Lang.Object getNextValue( ).
method public logical hasPrevKey().
method public class Progress.Lang.Object getPrevValue( ).
method public class Progress.Lang.Object getFirstValue( ).
method public class Progress.Lang.Object getLastValue( ).
```

I am not sure at this point of the utility of `keySet()`, `values()`, and `entrySet()`, so their implementation is being deferred. The last six methods are not a part of a traditional Java Map implementation, but seem like useful supplements based on Collections and thus have been added. All methods correspond closely to the functionality found for Collections, but with some name changes from Element to Value or Key as appropriate. Briefly, these are:

corresponding name. Briefly, this is:

`size()`	Returns the number of Key/Value pairs in the Map.
`isEmpty()`	Returns true if the Map has no Key/Value pairs.
`containsValue()`	Returns true if the Value is found in the Map.
`clearAll()`	Removes all Elements from the Map.
`keySet()`	Returns a SortedSet of the Keys in the Map.
`values()`	Returns a Collection of the Values in the Map.
`entrySet()`	Returns a SortedSet of type Entry, which is a simple class consisting of a single Key and Value.
`hasNextKey()`	Returns True if there is another unique Key beyond the current one.
`getNextValue()`	Returns the Value associated with the next unique Key beyond the current one; raises "No Such Element" exception and returns null if not available.
`hasPrevKey()`	Returns True if there is another unique Key prior to the current one.
`getNextValue()`	Returns Value associated with the next unique Key prior to the current one; raises "No Such Element" exception and returns null if not available.
`getFirstValue()`	Returns Value associated with the first unique Key; raises "No Such Element" exception and returns null if not available.
`getLastValue()`	Returns Value associated with the last unique Key; raises "No Such Element" exception and returns null if not available.

Note that methods that return Values, reposition the current key pointer, but methods that return Keys do not. Also, for ListMap classes, additional methods are provided for moving through the

Values associated with an identical Key; the Value returned by the methods above is always the first such Value associated with that Key.

In addtion to the common methods defined by the BaseMap interface ListMap and SetMap classes primarily contain methods which are identical except for the varying datatype of the Key. Where [Pre] indicates the Hungarian notation prefix and [Type] indicates the datatype of the key, these are:

```
method public logical containsKey(
    input m[Pre]_Key as [Type] ).
method public [Type] getFirstKey().
method public [Type] getLastKey().
method public [Type] getNextKey().
method public [Type] getPrevKey().
method public [Type] getCurrentKey().
method public class Progress.Lang.Object getValue(
    input m[Pre]_Key as [Type] ).
/*
method public void putAll(
    input m[Pre]_Map as class com.cintegrity.Lib.Collection.Map ).
*/
```

As for Collection classes the implementation of the more complex `putAll()` method is deferred in order to gain experience. The functionality of these methods is largely intuitive, to wit:

`containsKey()`	Returns True if the Key is contained within the Map; otherwise False.
`getFirstKey ()`	Returns the first unique Key; raises "No Such Element" exception and returns null if not available.
`getLastKey()`	Returns the last unique Key; raises "No Such Element" exception and returns null if not available.
`getNextKey()`	Returns the first unique Key beyond the current one; raises "No Such Element" exception and returns null if not available.
`getPrevKey()`	Returns the first unique Key prior to the current one; raises "No Such Element" exception and returns null if not available.
`getCurrentKey()`	Returns the current unique Key resulting from a prior action; raises "No Such Element" exception and returns null if not available.
`getValue()`	Returns the Value corresponding to the specified Key; raises "No Such Element" exception and returns null if not available.

As noted above, methods that return Values reposition the current key pointer, but methods that return Keys do not. Also, for ListMap classes, additional methods are provided for moving through the Values associated with an identical Key; the Value returned by the getValue is always the first such Value associated with that Key.

SetMap classes currently contain no methods not included in ListMap classes, but ListMap classes contain the following methods not included in SetMap classes:

```
method public integer countOfKey(
    input m[Pre]_Key as [Type] ).
method public integer getCurrentSeq().
method public integer numberOfKeys().
method public class Progress.Lang.Object getNextValueSameKey().
```

The functionality of these methods is as follows:

`countOfKey ()`	Returns the number of Values associated with the provided Key.
`getCurrentSeq()`	Returns the Sequence of the current Key/Value pair within the current key.
`numberOfKeys()`	Returns the number of unique Keys in the Map.
`getNextValueSameKey()`	Returns the Sequence of the current Key/Value pair within the current key.

In addition, there are two methods whose signatures differ between SetMap and List Map classes. The SetMap version is as follows:

```
method public class Progress.Lang.Object putValue(
    input m[Pre]_Key as [Type],
    input m[Pre]_Value as class Progress.Lang.Object ).
method public class Progress.Lang.Object removeValue(
    input m[Pre]_Key as [Type] ).
```

The ListMap version is:

```
method public void putValue(
    input m[Pre]_Key as [Type],
    input m[Pre]_Value as class Progress.Lang.Object ).
method public void removeValue(
    input m[Pre]_Key as [Type],
    input min_Seq as integer ).
```

Their functionality is as follows:

`putValue()`	Adds the Key/Value pair to the existing Map. For SetMap classes, if the Key exists, then the Value previously associated with that Key is returned.
`removeValue()`	Removes the Key/Value pair indicated by the provided Key (and Sequence, if a ListMap) from the current Map. For SetMap classes, the Value previously associated with that Key is returned.

SetMap classes have been implemented for the Key datatypes String, Integer, Datetime, and Object. ListMap classes have been implemented for those datatypes as well as Date and Decimal. Other datatypes were considered unlikely, but could be implemented very quickly in the existing code, should the need arise.

Entry

Entry is an interface that defines a single key/value pair in a Map. It has methods:

```
method public class Progress.Lang.Object getKey().
method public class Progress.Lang.Object getValue().
method public class Progress.Lang.Object setValue().
```

Entry has not been implemented in the current release because the methods that use it have not been implemented.

Summary Comparison of Concrete Classes

With the deletions for classes which are not needed because they are defined by an implementation we will not use, and the addition of Map classes covering various key datatypes and having both

unique key and non-unique key forms, we are left with SortedSet, List, and a variety of Map as classes as the only concrete classes. SortedSet and List are Collections of objects on their own; Map classes are collections of key/value pairs. SortedSet implies no duplicate objects; List does not. XxSetMap has no duplicate Keys, but can have duplicate value objects; XxListMap can have duplicate keys as well. See table at end for detailed list of all classes.

Synchronization

Synchronization will not be addressed in early implementations of this library since Progress is not currently multi-threaded.

Unmodifiable Wrappers

Java Collection classes provide methods to return an unmodifiable wrapper, i.e., a class of the same type *backed by* the original class, but which is effectively read-only. It has not currently been determined how or when to implement these.

Collection Utility Methods

In addition to the methods defined above, Java Collection classes include useful utility static methods including:

```
method public class Progress.Lang.Object min().
method public class Progress.Lang.Object minBy(
    input mob_Comparator as class com.cintegrity.util.Comparator ).
method public class Progress.Lang.Object max().
method public class Progress.Lang.Object maxBy(
    input mob_Comparator as class com.cintegrity.util.Comparator ).
```

In the Java implementation, all of the min and max methods take a Collection object as a first argument. When and if we implement these methods, they are likely to be part of the Collection or Map itself. Because of the various FirstKey and LastKey methods, they may be largely redundant, but that will be further evaluated in future releases.

There are no immediate plans to implement the following methods:

```
method public void reverse().
method public void shuffle().
method public void fill(
    input mob_Element as class Progress.Lang.Object).
method public void copy(
    input mob_Source as class Progress.Lang.Object,
    input mob_Destination as class Progress.Lang.Object ).
method public class Progress.Lang.Object nCopies(
    input min_Count as integer,
    input mob_Destination as class Progress.Lang.Object ).
method public class Progress.Lang.Object singleton(
    input mob_Element as class Progress.Lang.Object).
method public com.cintegrity.util.List singletonList(
    input mob_Element as class Progress.Lang.Object).
method public class com.cintegrity.util.Map singletonMap(
    input mob_Key as class Progress.Lang.Object,
    input mob_Value as class Progress.Lang.Object ).
method public void sort(
    input mob_List as class com.cintegrity.util.List).
method public void sortBy(
    input mob_List as class com.cintegrity.util.List).
    input mob_Comparator as class com.cintegrity.util.Comparator).
```

```
method public integer binarySearch(
    input mob_List as class com.cintegrity.util.List).
    input mob_Key as class com.cintegrity.util.Key).
method public integer binarySearchBy(
    input mob_List as class com.cintegrity.util.List).
    input mob_Key as class com.cintegrity.util.Key,
    input mob_Comparator as class com.cintegrity.util.Comparator).
```

PostScript

This document corresponds to the indicated release level of this set of classes. It is intended to develop the classes further based on experience, feedback, and obvious areas of additional opportunity. As new releases are created, this document will be revised accordingly and reissued.

Summary of Java Versus CI ABL Collection Classes

Java Class	ABL Class	Discussion
Collection	Collection	ABL version incorporates methods from Iterator
Iterator		Incorporated in Collection because of use of temp-tables
ListIterator		Incorporated in Collection because of use of temp-tables; provided for all Collection Classes
Comparable		Not implemented as it becomes trivial when temp-tables are used for implementation.
Comparator		Not implemented as it becomes trivial when temp-tables are used for implementation.
Set		Not implemented since use of temp-tables will provide SortedSet in all cases and Set is a reserved word.
SortedSet	SortedSet	Implemented as a concrete class.
HashSet		Not implemented since the concrete implementation uses temp-tables.
TreeSet		Not implemented since the concrete implementation uses temp-tables.
List	List	Implemented as a concrete class.
ArrayList		Not implemented since the concrete implementation uses temp-tables.
LinkedList		Not implemented since the concrete implementation uses temp-tables.
Map	Map	Not implemented because Map is a reserved word. Methods from Map are found in BaseMap
SortedMap		Folded into Map because of the use of temp-tables
TreeMap		Not implemented since the concrete implementation uses temp-tables.
HashMap		Not implemented since the concrete implementation uses temp-tables.
WeakHashMap		Not implemented since the concrete implementation uses temp-tables.
	BaseMap	Contains methods from Map
	*SetMap	Logical group of classes that implement unique key maps as are found in Java.
	*ListMap	C Logical group of classes that provide “List-like” Maps, i.e., no constraint on uniqueness of keys.
	StringObjectSetMap	Concrete class of SetMap with character key and object value.
	IntegerObjectSetMap	Concrete class of SetMap with integer key and object value.
	DecimalObjectSetMap	Concrete class of SetMap with decimal key and object value (deferred).
	DateObjectSetMap	Concrete class of SetMap with date key and object value (deferred).
	DatetimeObjectSetMap	Concrete class of SetMap with datetime key and object

Java Class	ABL Class	Discussion
		value.
	ObjectObjectSetMap	Concrete class of SetMap with object key and object value.
	StringObjectListMap	Concrete class of ListMap with character key and object value.
	IntegerObjectListMap	Concrete class of ListMap with integer key and object value.
	DecimalObjectListMap	Concrete class of ListMap with decimal key and object value.
	DateObjectListMap	Concrete class of ListMap with date key and object value.
	DatetimeObjectListMap	Concrete class of ListMap with datetime key and object value.
	ObjectObjectListMap	Concrete class of ListMap with object key and object value.

While not initially planned for implementation, this naming scheme allows for development of Map classes in which the value is simple instead of an object, e.g., IntegerStringListMap.