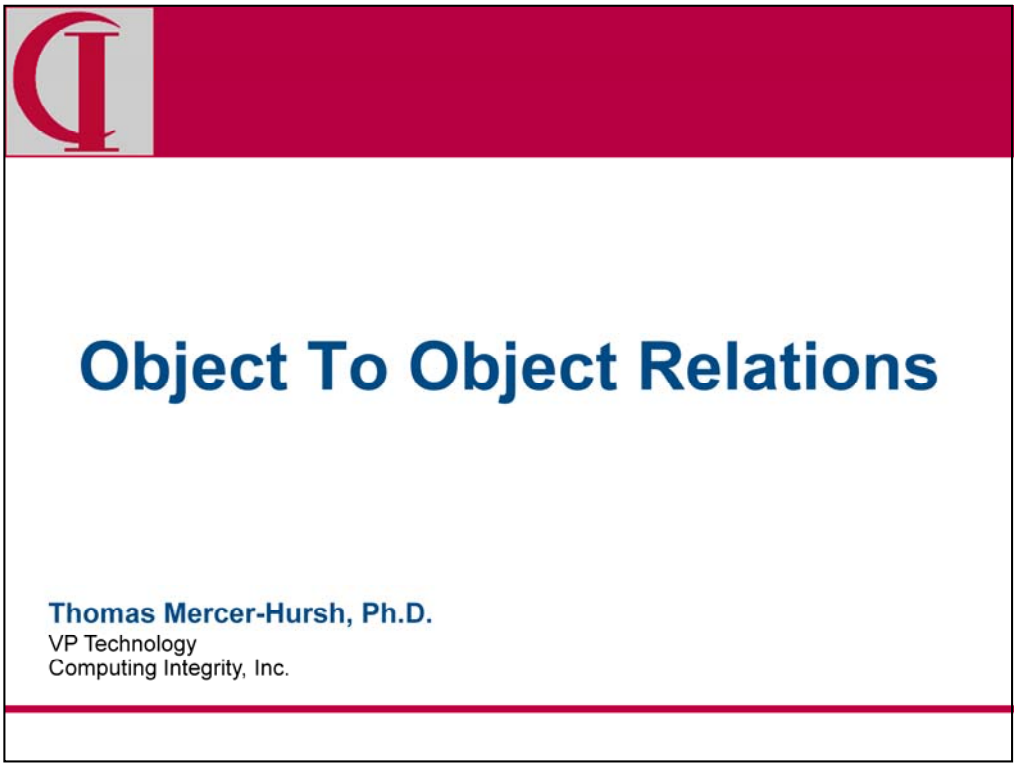
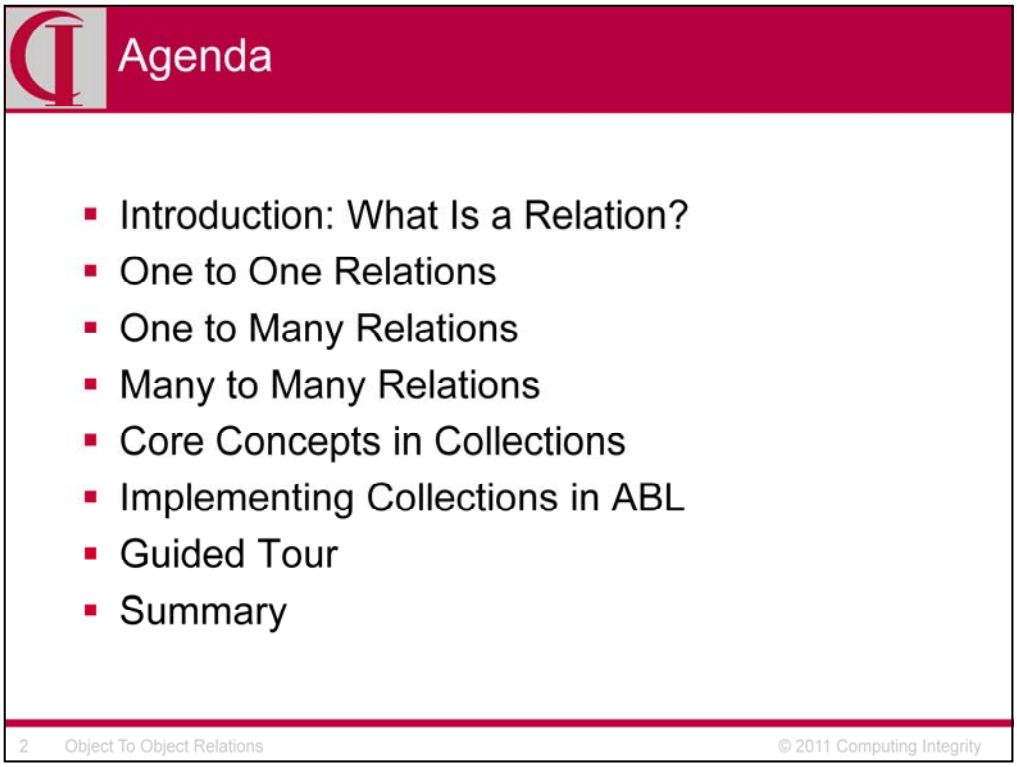
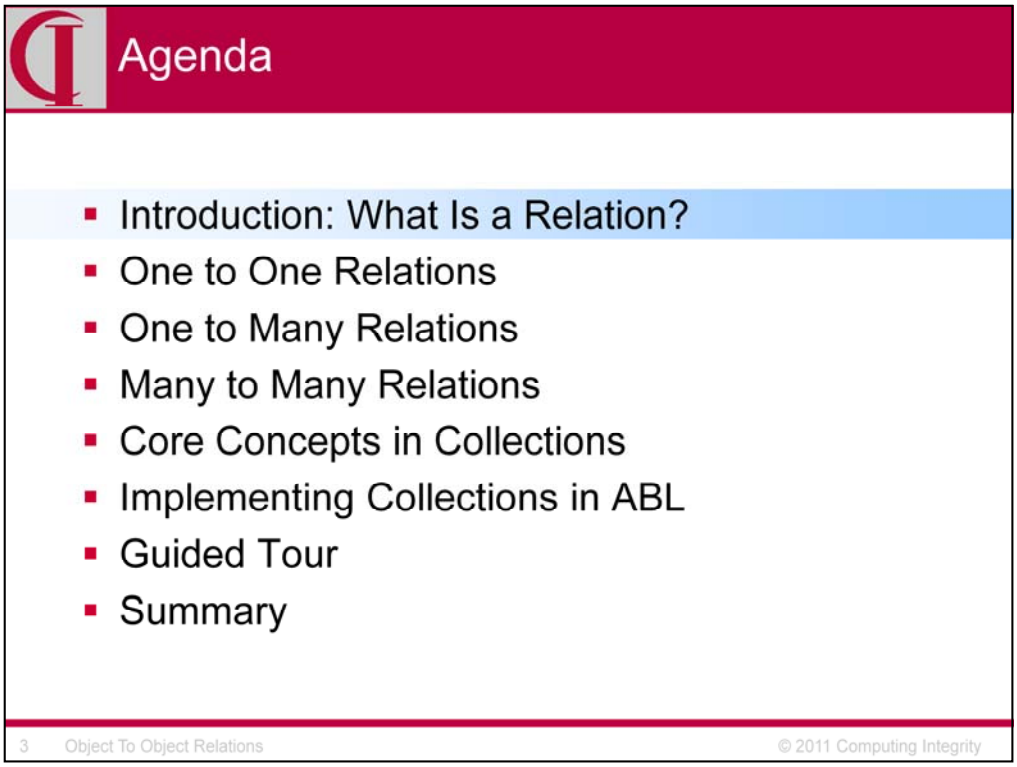


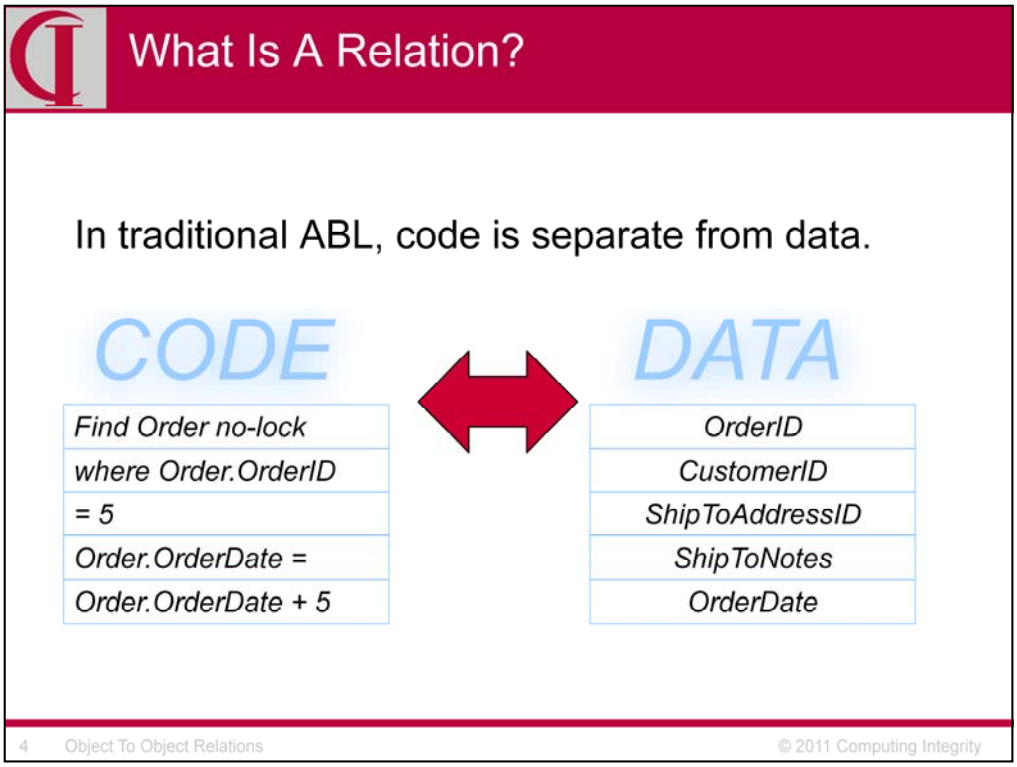
Let me begin by introducing myself. I began working with Progress in 1984 and I have been a Progress Application Partner since 1986. For many years I was the architect and chief developer for our ERP application. In recent years, I have refocused on the problems of transforming and modernizing legacy ABL applications. Object Orientation is widely accepted as a preferred paradigm for developing complex applications by much of the programming world and now that OO features are now in ABL, I and others have been exploring the benefits of using OO in ABL.



So, here's our agenda for today. First we are going to talk a bit about what a relation is and how it contrasts with what one does in traditional ABL. Then we will talk about different types of relations according to the number of objects on each side. Next, we will look at core concepts in collections. Finally, we will look at options for implementing collections in ABL and look at some actual objects.

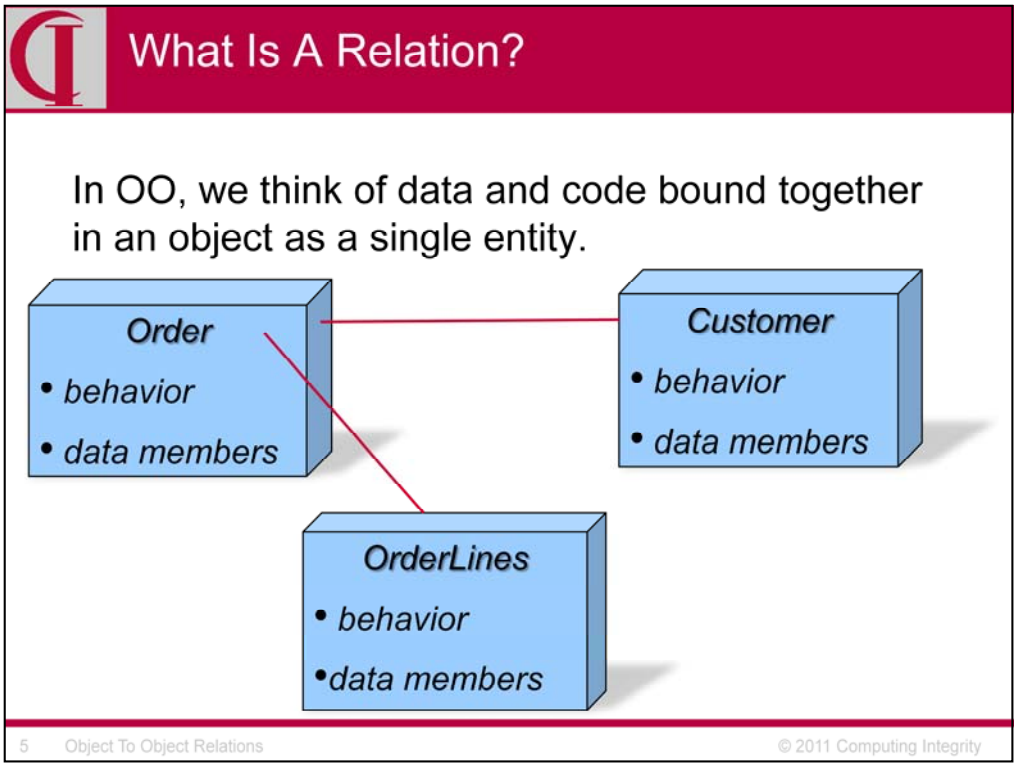


First, let's talk about what we mean when we say "relation".



In traditional ABL, we think of code as a separate thing from data.

Much ABL data we think of in terms of a relational model, i.e., values arranged in tuples, located by means of keys.

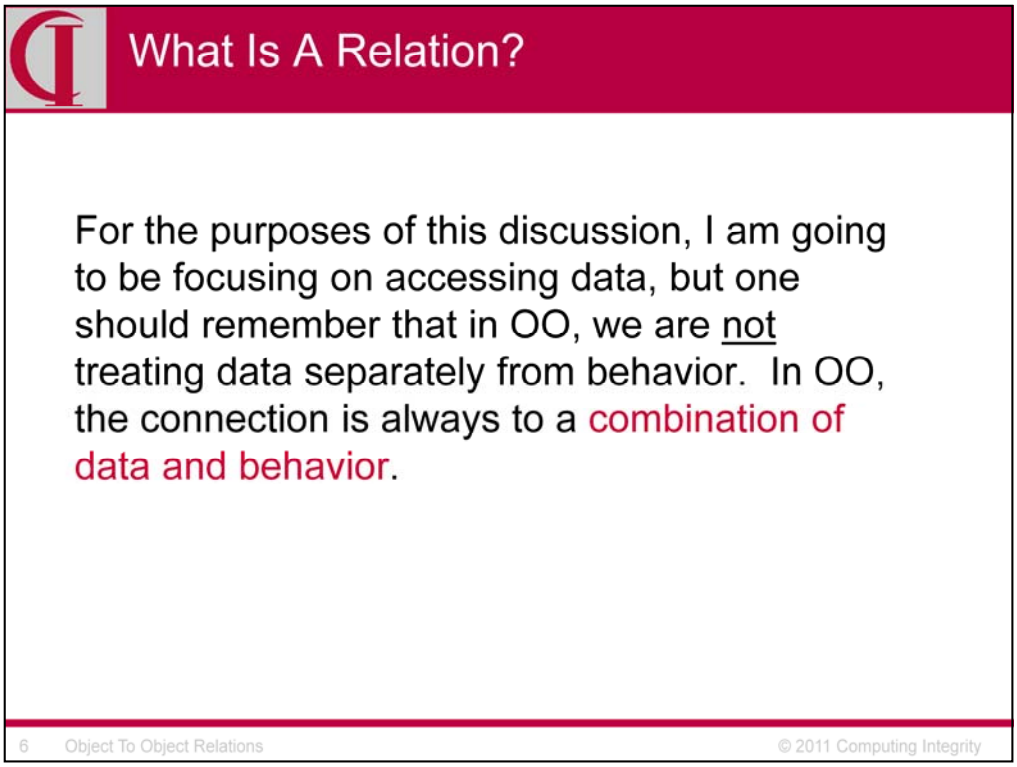


In OO, we think of data and code bound together in an object as a single entity.

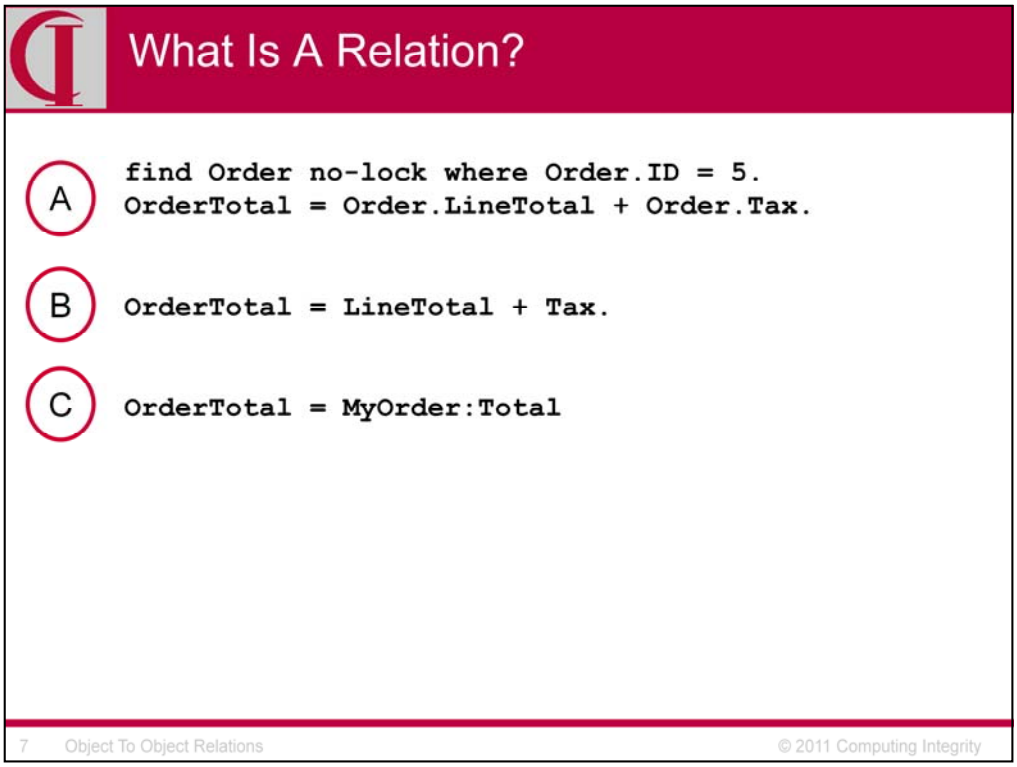
Thus, in OO, we connect Object to Object, not code to data.

There is no Customer key in Order. Rather, there is a variable which is an actual Customer object.

Connections are direct references, not something one looks up.



For the purposes of this discussion, I am going to be focusing on accessing data, but one should remember that in OO, we are not treating data separately from behavior. In OO, the connection is always to a combination of data and behavior.

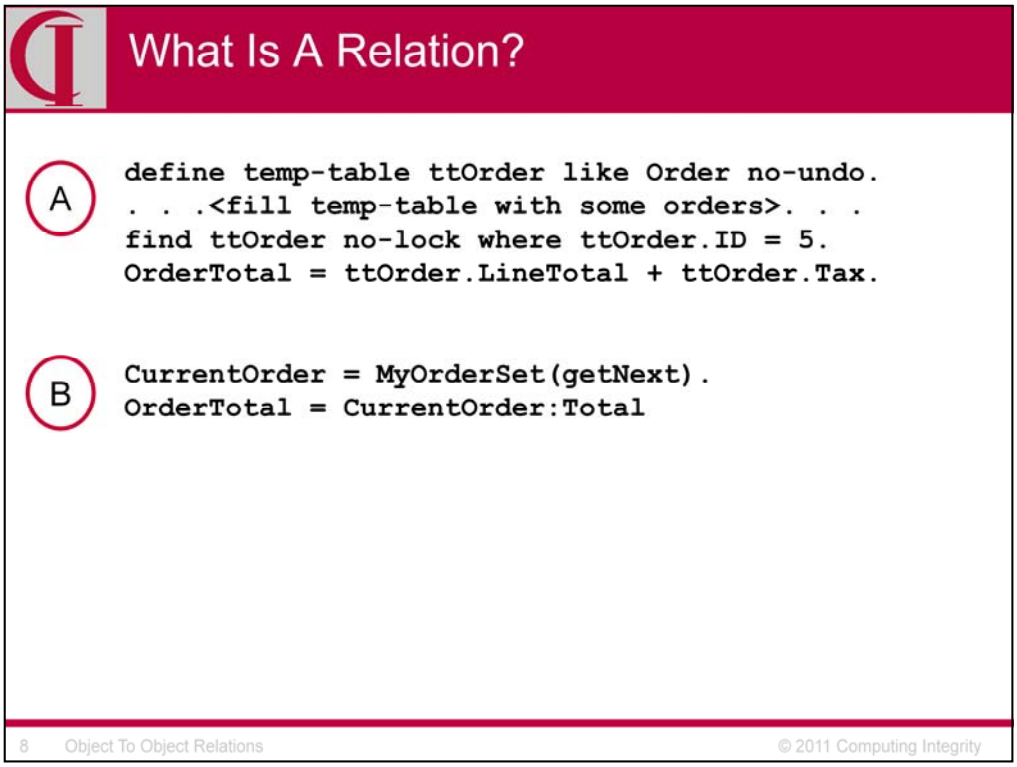


Compare these three code fragments.

In code fragment A, in order to obtain an Order total in the way we would in traditional ABL … in a remarkably simple system … we have to find the order of interest and then operate on its properties. This Order might be coming from a database table or a local temp-table. This code might appear anywhere we need the order total. Functionally, we have to do something to find the right set of order data and then do our computation.

In code fragment B we have the code we would expect in an order object. I.e., the properties are the object's own properties so there is nothing to find.

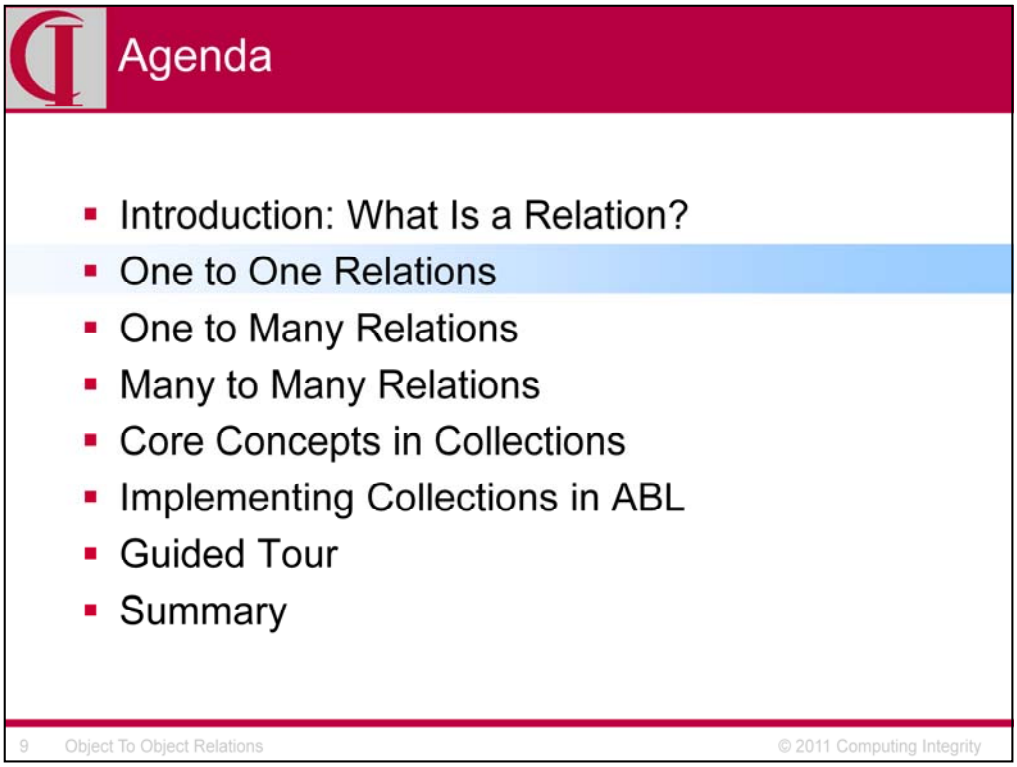
In code fragment C we are looking at things from the same perspective as A, i.e., from something outside the Order using the Order. Since the Order knows how to compute its total, we don't do the computation, but ask MyOrder for the correct value. We don't find MyOrder, but rather any one of a number of different possible processes has lead to us having a direct connection to it. We just reference it.



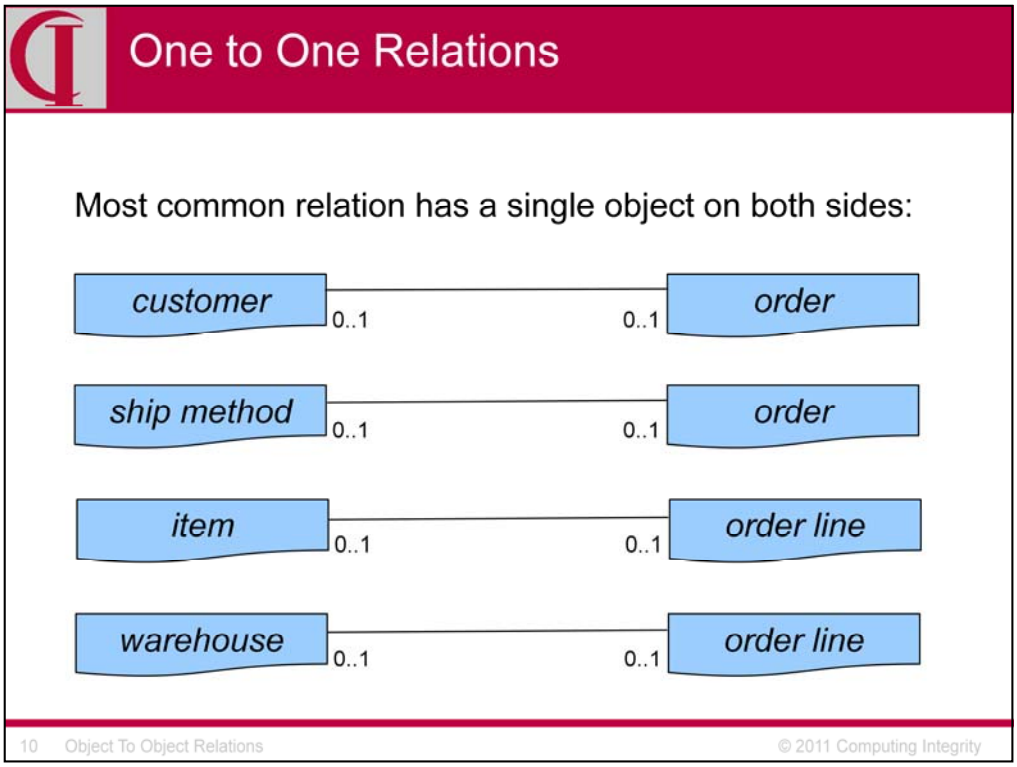
The contrast is perhaps more obvious if look at the example of being connected to a set of orders.

In code fragment A, we define a temp-table for orders … by the way, in practice I never use LIKE, but here it makes the sample shorter … and then fill that temp-table with some set of orders according to some criteria. We then find one of those orders, or possibly define a query and work through them sequentially. We compute the total in the same way by reference to the current buffer, a buffer whose identity is determined by a key, even if we are moving sequentially.

In code fragment B we have an object MyOrderSet which contains a collection of orders obtained in some way, most probably not by the current object. We will talk about collections more in a little while. It has operations like getNext that return an object of type Order. Having obtained one, we get the total by a direct request of the object like in the prior slide.

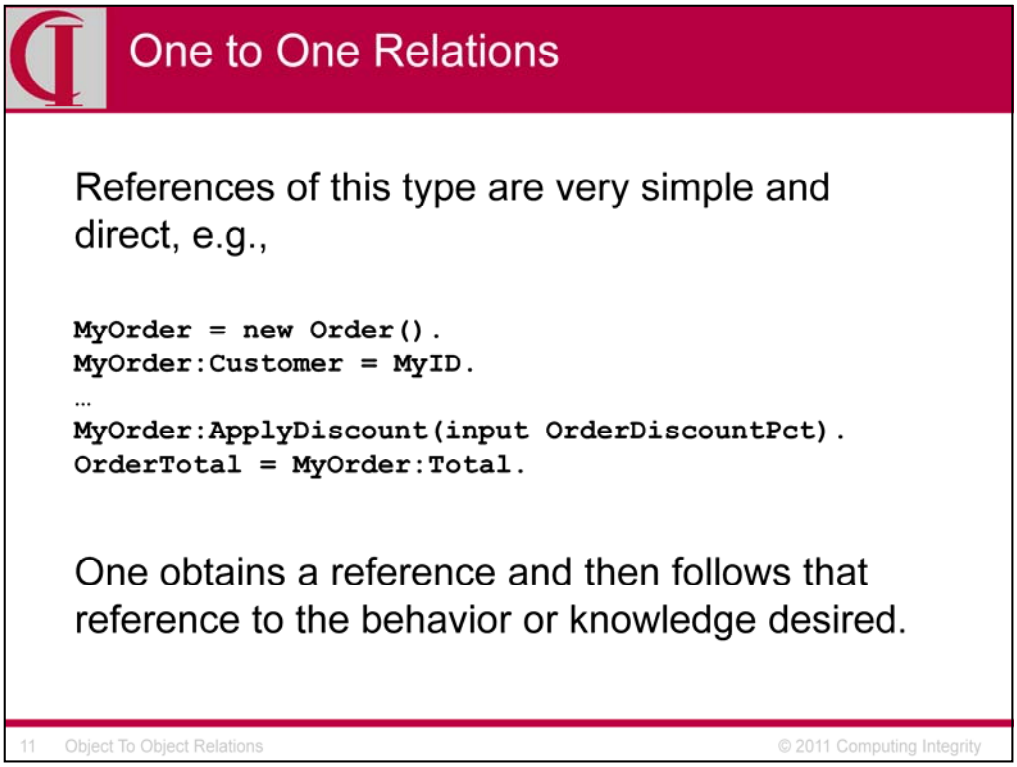


There are three possibilities for the number of objects on each side of a relation – one on each side, i.e., one to one; one on one side and more than one on the other, i.e., one to many; or multiple objects on both sides; i.e., many to many. Let's look at these each in turn.



The most common type of relation has a single object on both sides. Examples include:

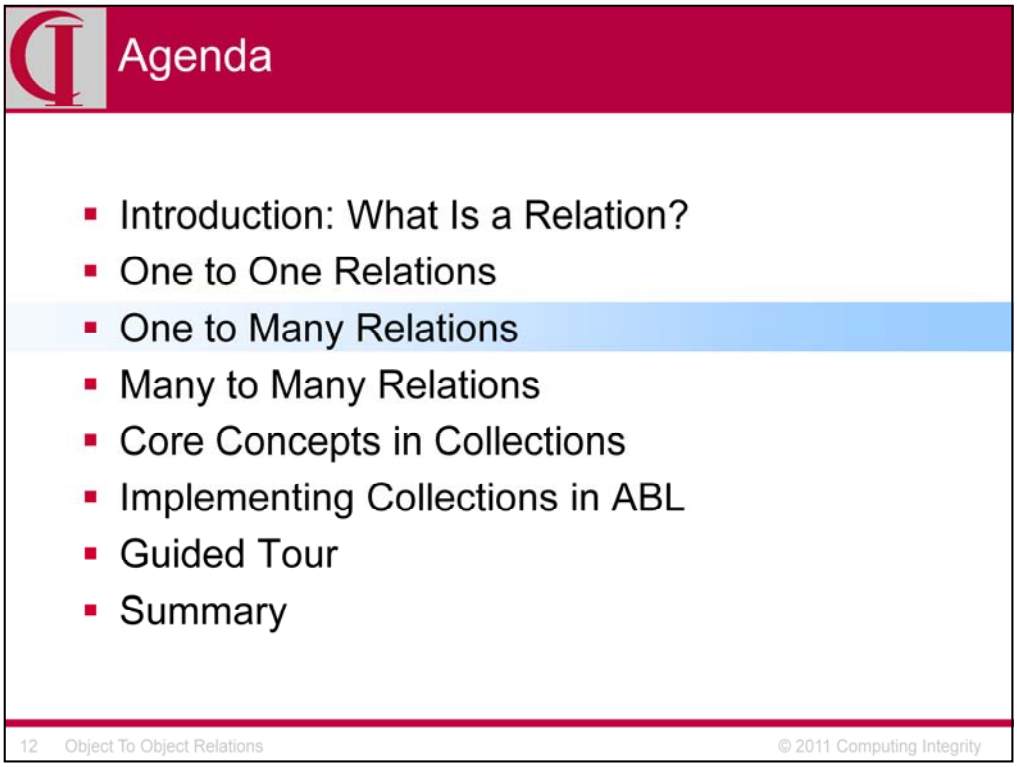
- The customer of an order.
- The shipping method of the order.
- The item of the order line.
- The warehouse for the order line.



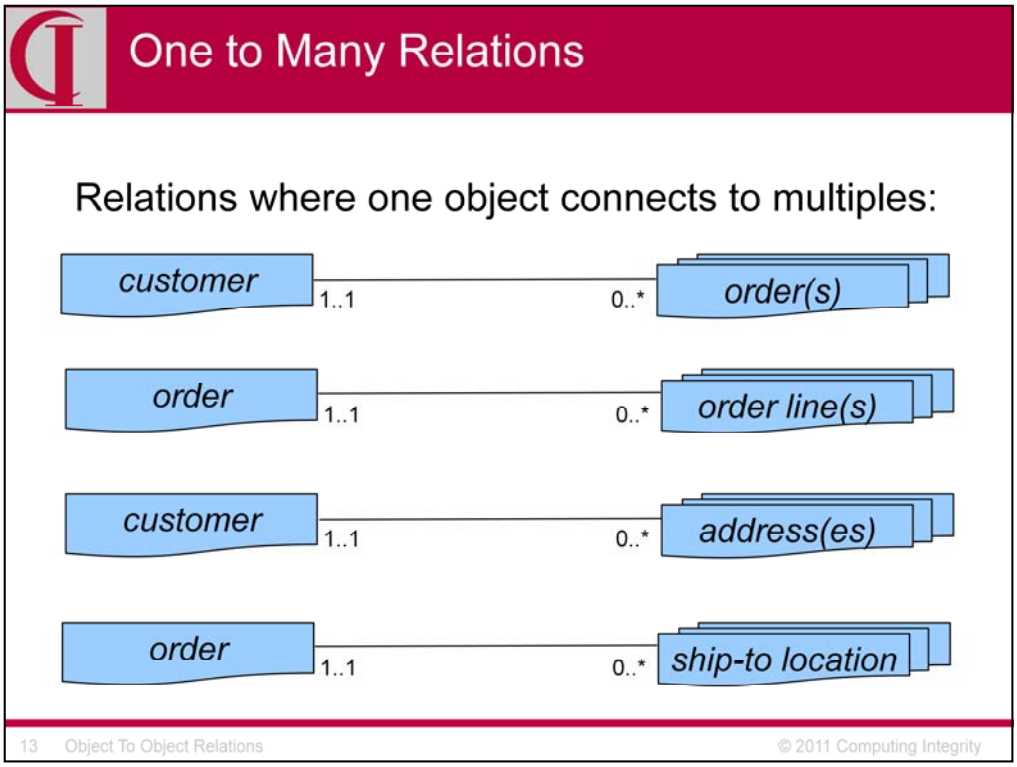
References of this type are very simple and direct.

In the first line we are setting a local variable to the reference for an new object. In the second, we are assigning a value to a property of that object. Next, we give the object a value and use it in calculating a method. Finally, we retrieve the value of a property.

In each case we obtain a reference to the object and use that reference to directly refer to the knowledge or behavior in the object.



Now let's look and One to Many Relations.

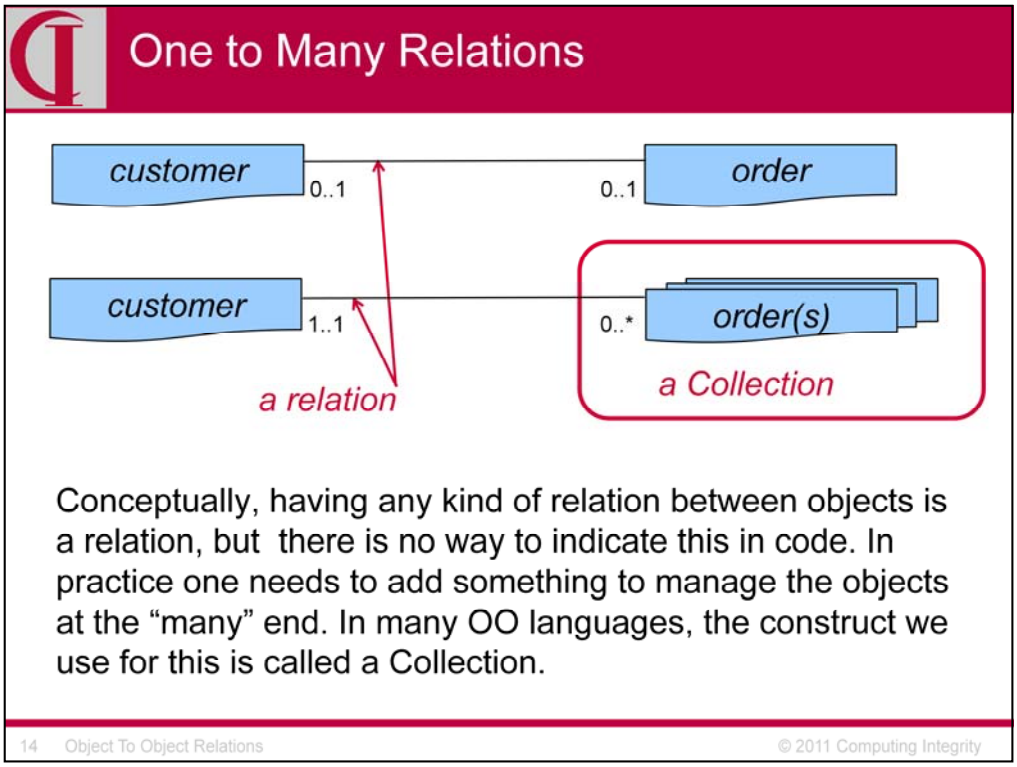


Relations in which there is one object on one side of the relation and multiple of the same object on the other side are also quite common. Examples include:

- The orders of a customer.
- The lines of an order.
- The addresses of a customer.
- The ship-to locations for an order.

Note that the issue is not whether there is actually more than one object at the multiple end of the relation, but whether there might be more than one. Thus, we might have an order that has only one line, but we have to handle the relation as if there were more than one since there sometimes will be more than one and we want to handle it uniformly. Note that there are cases where only one is typical, e.g., the shipping address for an order, and cases where one is rare, e.g., orders per customer in a business to business company.

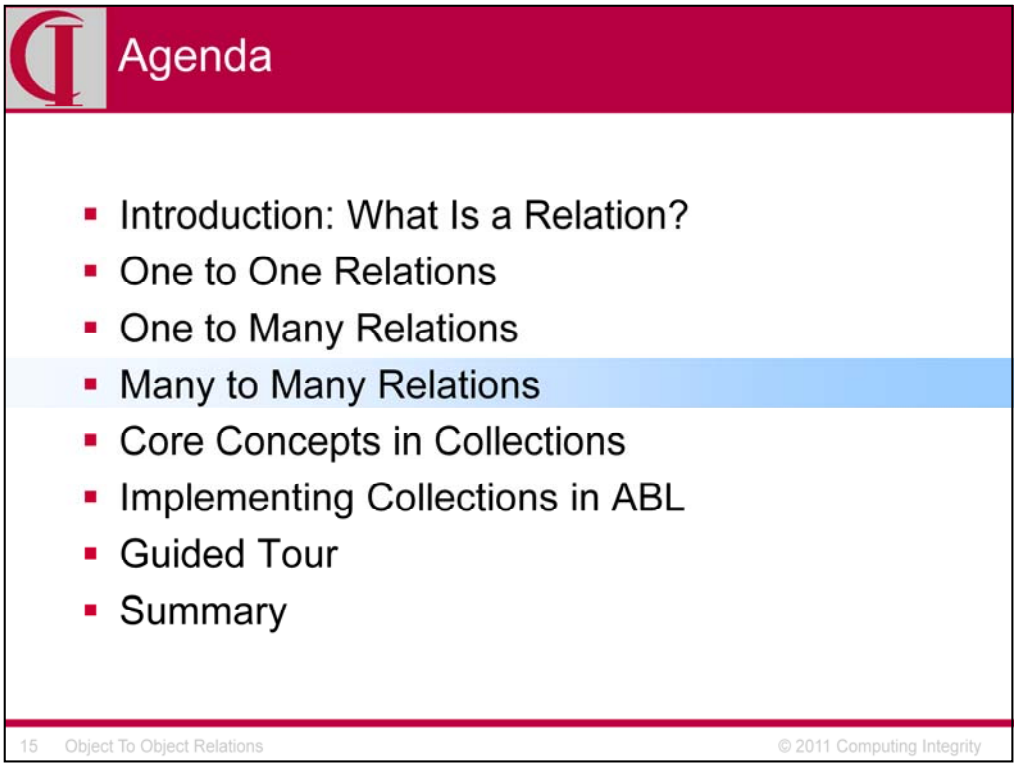
This is the sort of data that one would typically represent as a temp-table in traditional ABL. There are some who advocate continuing to use temp-tables for this purpose, but then one loses the encapsulation of knowledge and behavior which is fundamental to OO. In OO terms, what one expects at the multiple end of such a relation is a set of objects, not a temp-table and some associated behavior elsewhere. The Progress Professional Services CloudPoint model takes a sort of middle ground in which the data is kept in temp-tables, but these are manifested one at a time into single entity objects as they are accessed.



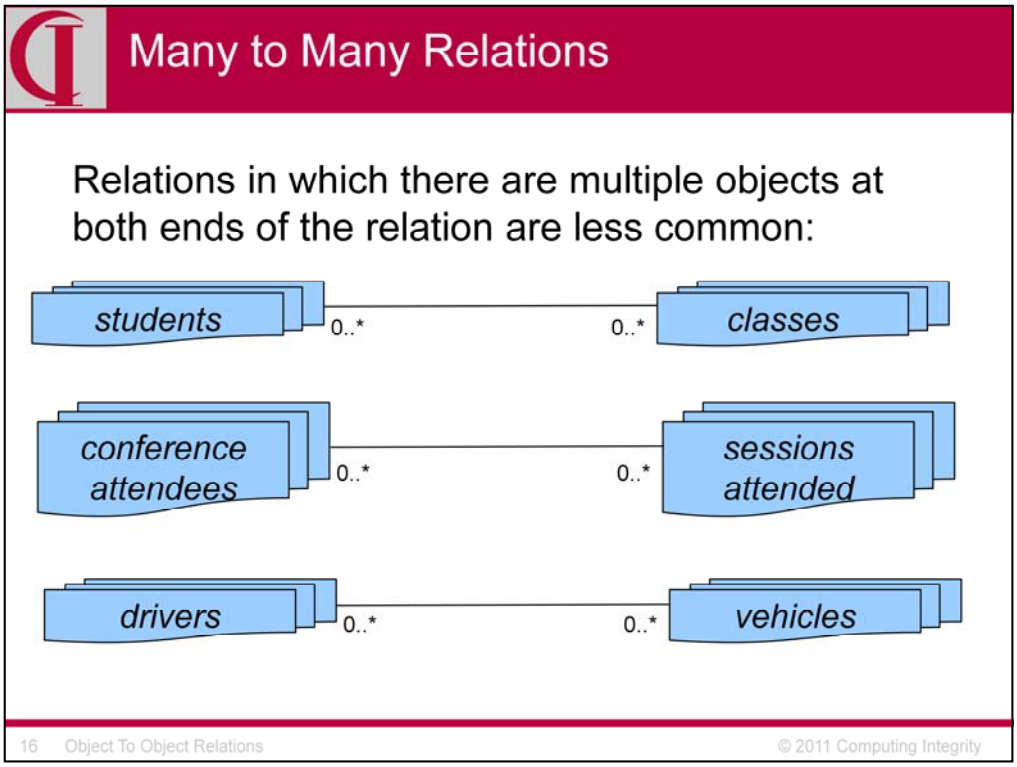
Conceptually, one to many relations are just a flavor of relations, but in practice one needs to add something to manage the objects at the many end.

In many OO languages, the construct we use for this is called a Collection .

We will explore the issues involved in designing a collection framework later.



Finally, let's take a look at many to many relations.



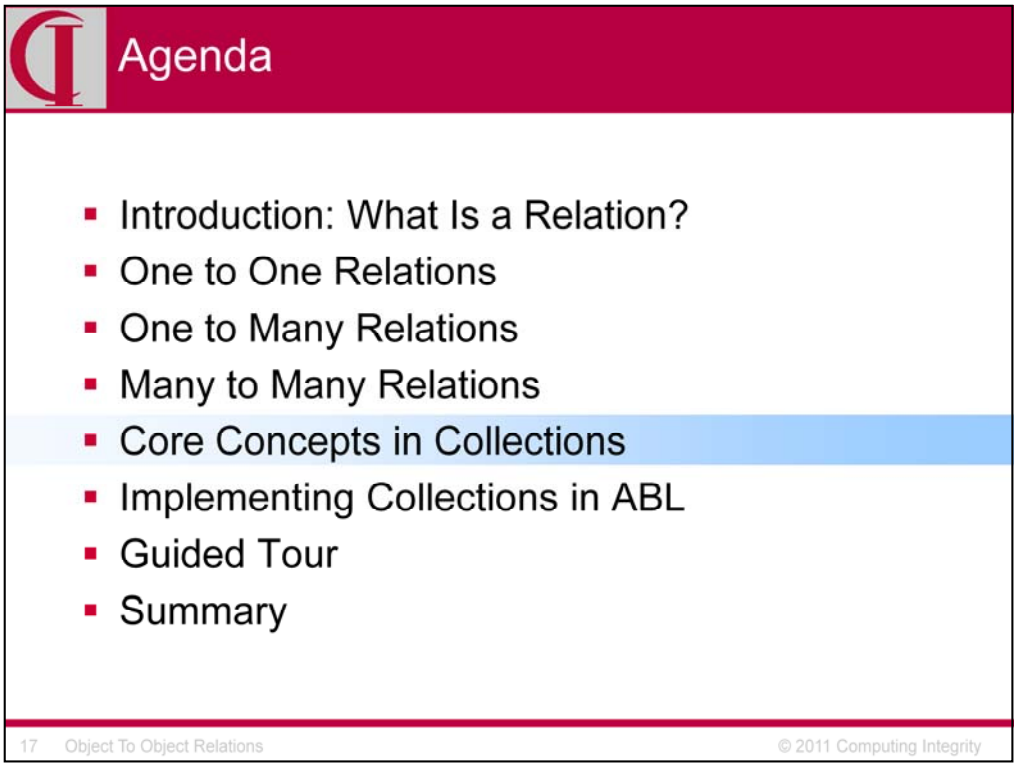
Relations in which there are multiple objects at both ends of the relation are less common, but still important.

Examples include:

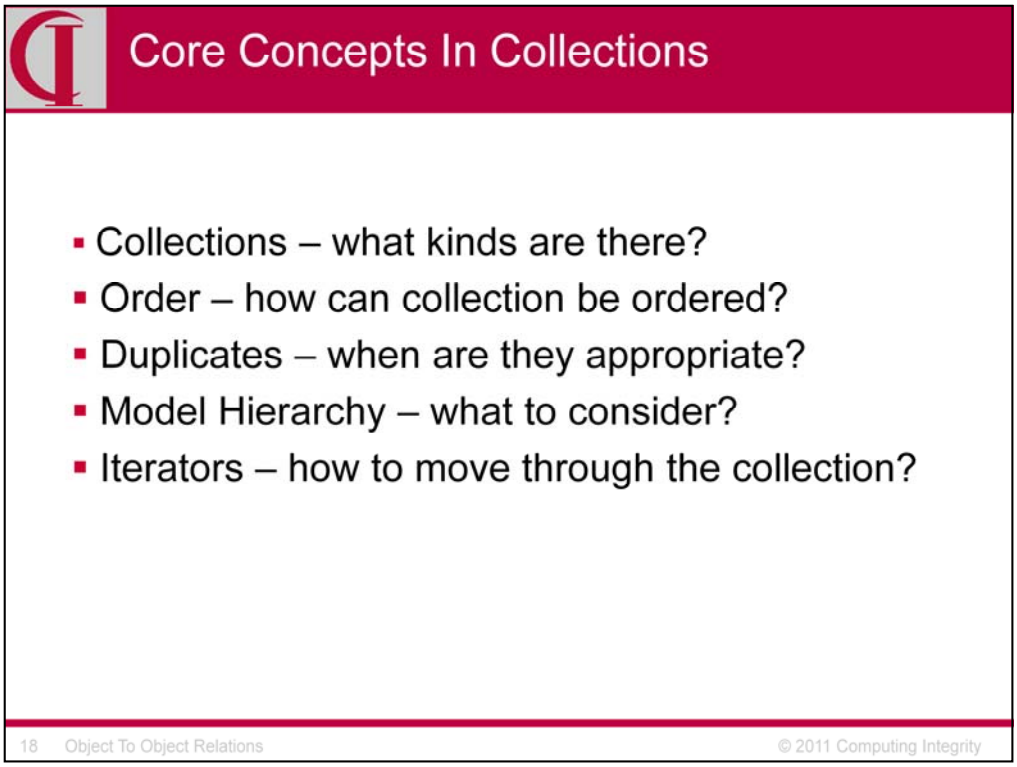
- Students who are attending current classes.
- Attendees at a conference and the sessions they attend.
- Drivers and the vehicles they drive.

Note that inherent in any many to many relationship are a whole bunch of one to many relationships. Thus, any one student will have a one to many relationship to his or her classes and any one class will have a one to many relationship between the class and the students attending that class. It is when we take all students and all classes that we get the many to many relationship. Thus, in normal processing, it is frequent that an inherently many to many relationship will be handled as a one to many relationship since we are focusing on one member of the group on one side of the relation at a time. I.e., either we will be considering one student a time or one class at a time.

Consequently, we are going to focus on one to many relationships and how we implement them for the rest of this presentation.

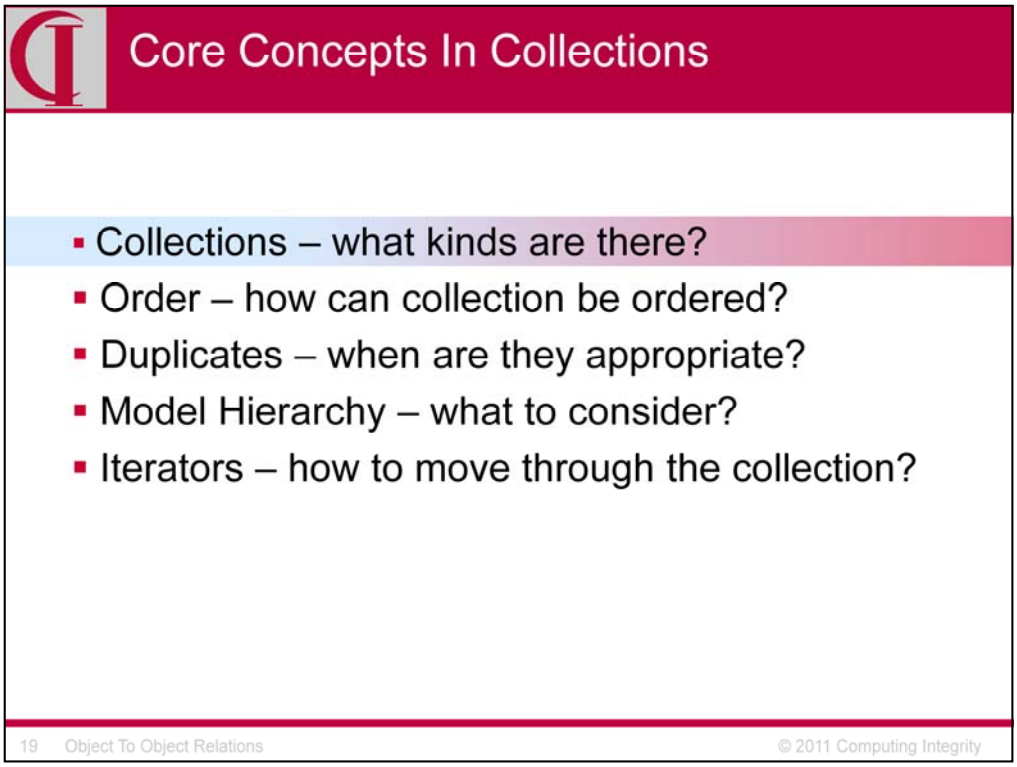


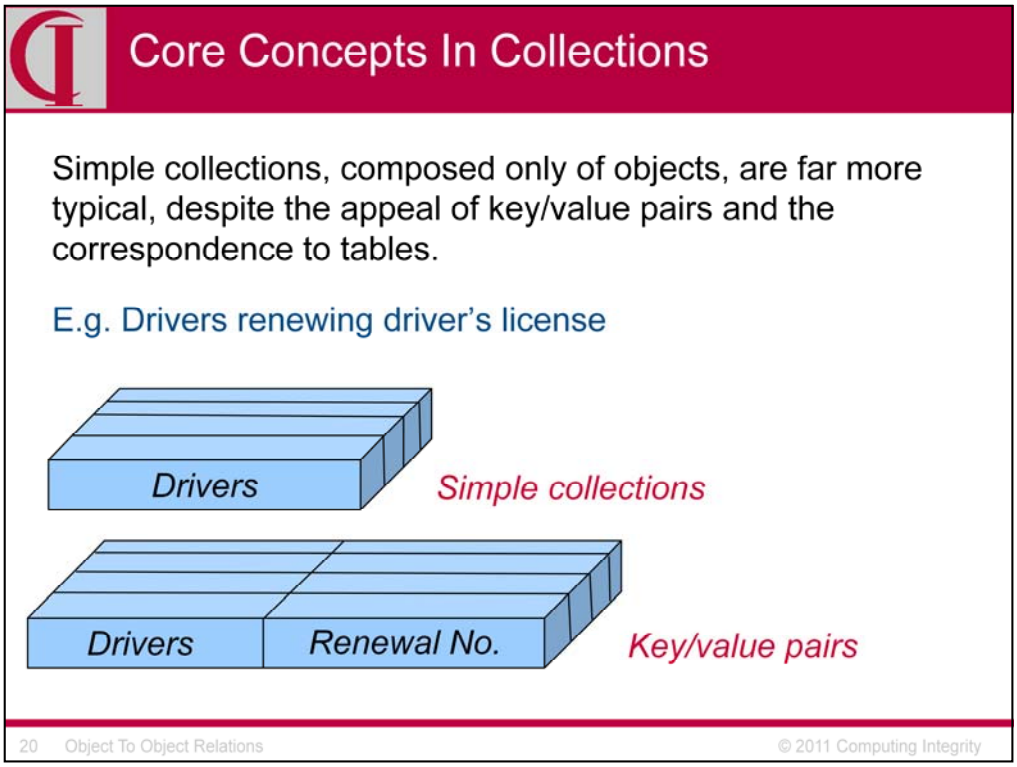
First, let's take a look at the core concepts we need to consider in designing a set of classes to provide us with collection management.



These are the core concepts which we need to consider in designing collection classes for ABL

- Collections – what kinds are there?
- Order – how can collection be ordered?
- Duplicates – when are they appropriate?
- Model Hierarchy – what to consider?
- Iterators – how to move through the collection?


Core Concepts In Collections
Collections – what kinds are there?
Order – how can collection be ordered?
Duplicates – when are they appropriate?
Model Hierarchy – what to consider?
Iterators – how to move through the collection?
19 Object To Object Relations
© 2011 Computing Integrity



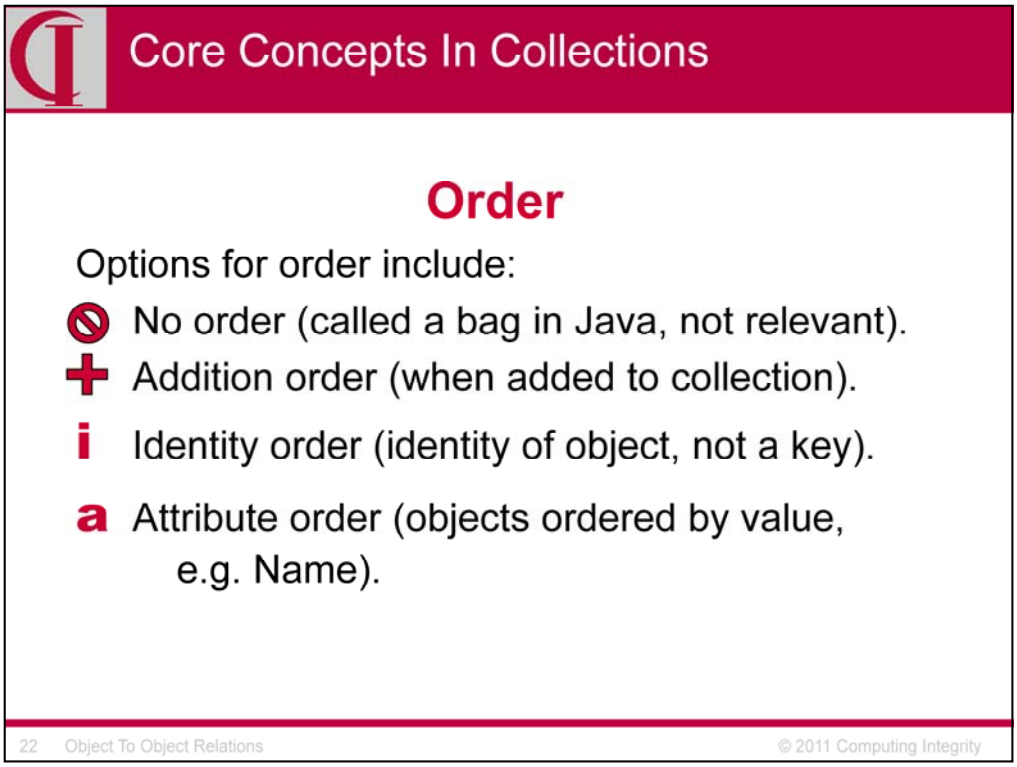
Generally, one thinks of two kinds of collections:

- Simple collections composed only of objects.
- Collections composed of key/value pairs.

Simple collections are far more typical in actual use in OO code, despite the familiarity of key/value pairs and the parallel to tables.

Core Concepts In Collections

- Collections – simple and key/value pairs
- Order – how can collection be ordered?
- Duplicates – when are they appropriate?
- Model Hierarchy – what to consider?
- Iterators – how to move through the collection?



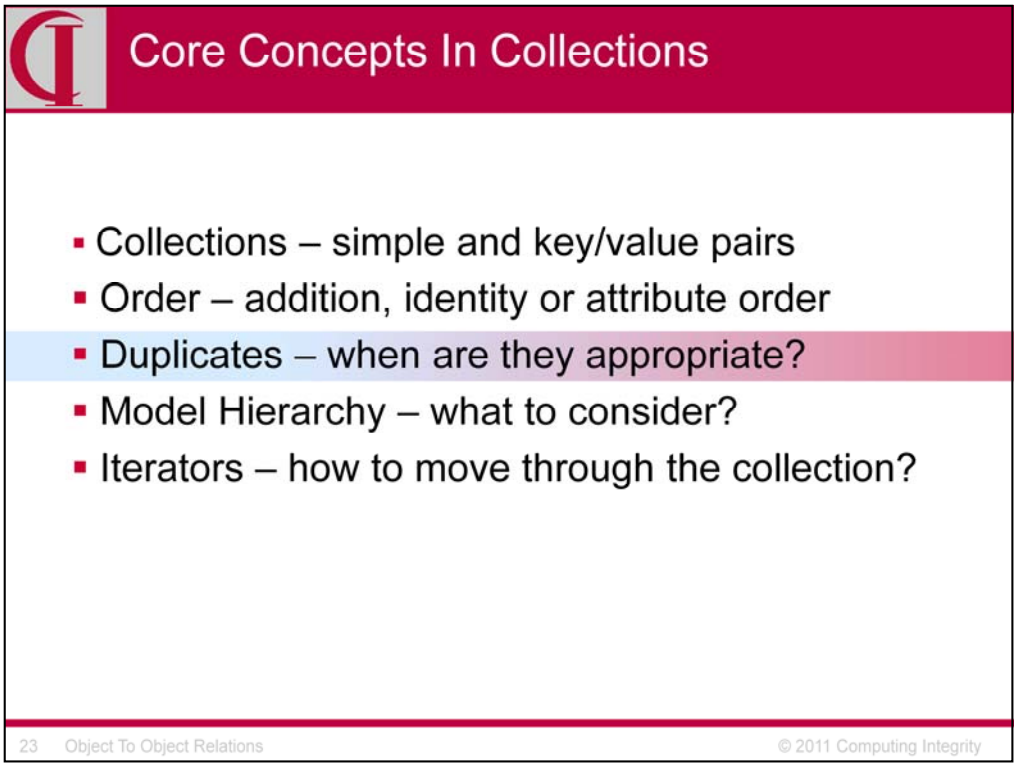
One issue which we should consider in designing collection classes is order, i.e., the sequence in which members will be delivered in response to “get next”. Options for order include:

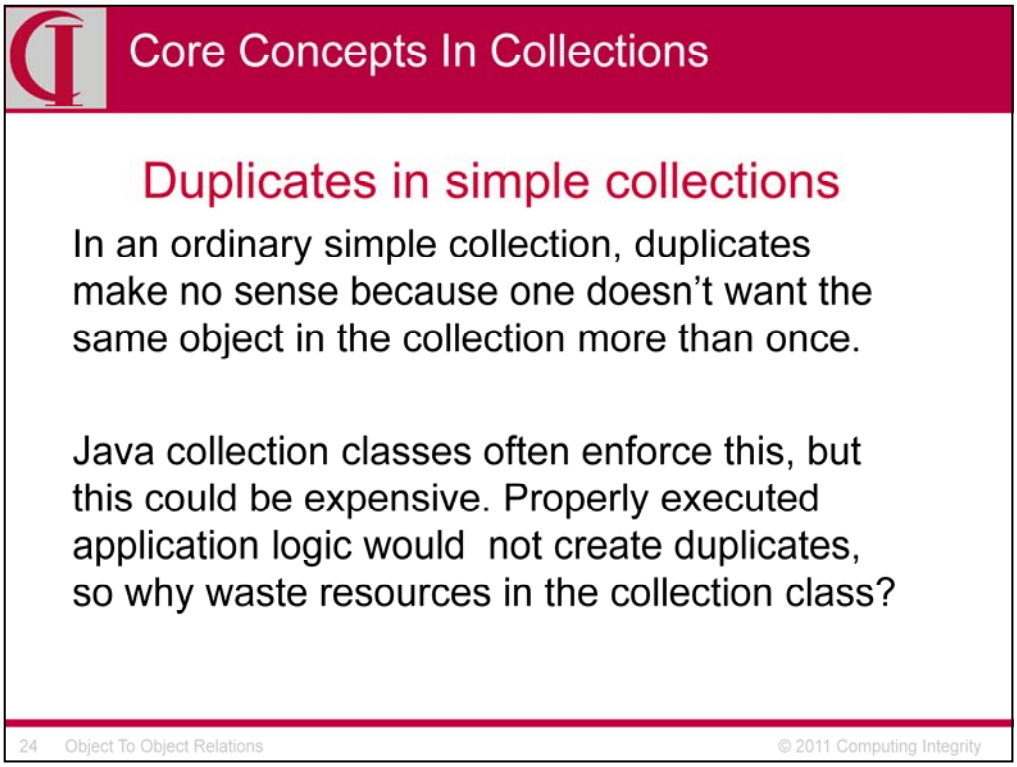
No order
Order is not predictable in any way. Called a bag in Java. There is no apparent value to such a collection so we will omit it here.

Addition order
Ordered only by the order in which the objects are added to the set. Processing typically proceeds sequentially through all members. This is the most common requirement. To those of us used to relying on keys, it may be surprising that order often doesn’t matter, but any operation which is just going to process all members is often one in which order doesn’t matter. Obviously, in presentation, order might matter, but that is a special context. Note that it is often possible to control addition order and thus to determine access order.

Identity order
Identity order is ordering by the identity of the object, not a key. This is a less common requirement, but this need arises when it is desirable to find an object based on its identity.

Attribute order
In attribute order, objects are ordered according to some other value, typically an attribute of the objects like Name. While this seems common from the perspective of traditional ABL, other than UI, it is actually not that common in most OO.


Core Concepts In Collections
Collections – simple and key/value pairs
Order – addition, identity or attribute order
Duplicates – when are they appropriate?
Model Hierarchy – what to consider?
Iterators – how to move through the collection?
23 Object To Object Relations
© 2011 Computing Integrity

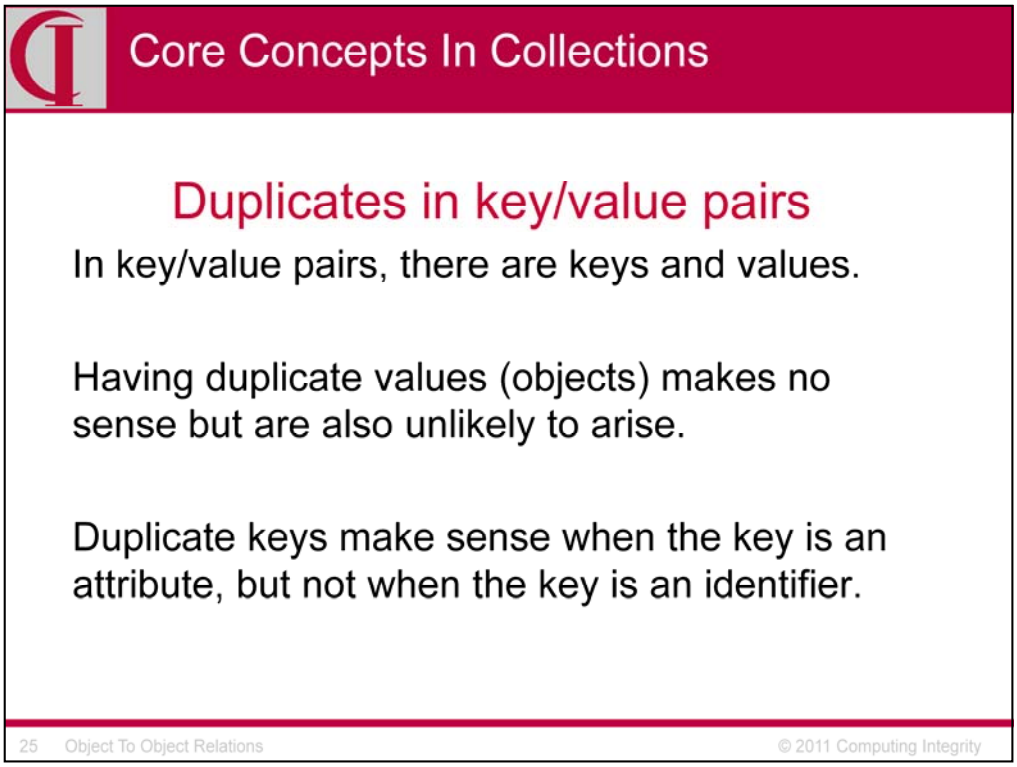


Another issue which we should consider in designing collection classes is duplicates.

In an ordinary simple collection, one would not allow duplicates because it makes no sense to have the same object in the collection more than once.

In a key/value pair collection, there is a question about whether one allows duplicate objects with different keys or no duplicate objects. If the key is an identifier, e.g., something like an order number, then we would not allow duplicates. If it is a non-unique attribute, there are cases where we might allow duplicate keys as well as cases where we might even allow duplicate objects if an object has more than one value for the same attribute.

Note that, while a mixture of unique and non-unique keys on a set of data sounds like bread and butter traditional ABL, with an OO orientation we are not thinking about all possible relationships and structures at each time but about the relationship which is relevant in the current context. Thus, typically one order and one policy about duplicates will apply to each circumstance and a simple solution for the purpose will get used in place of a very generalized solution.

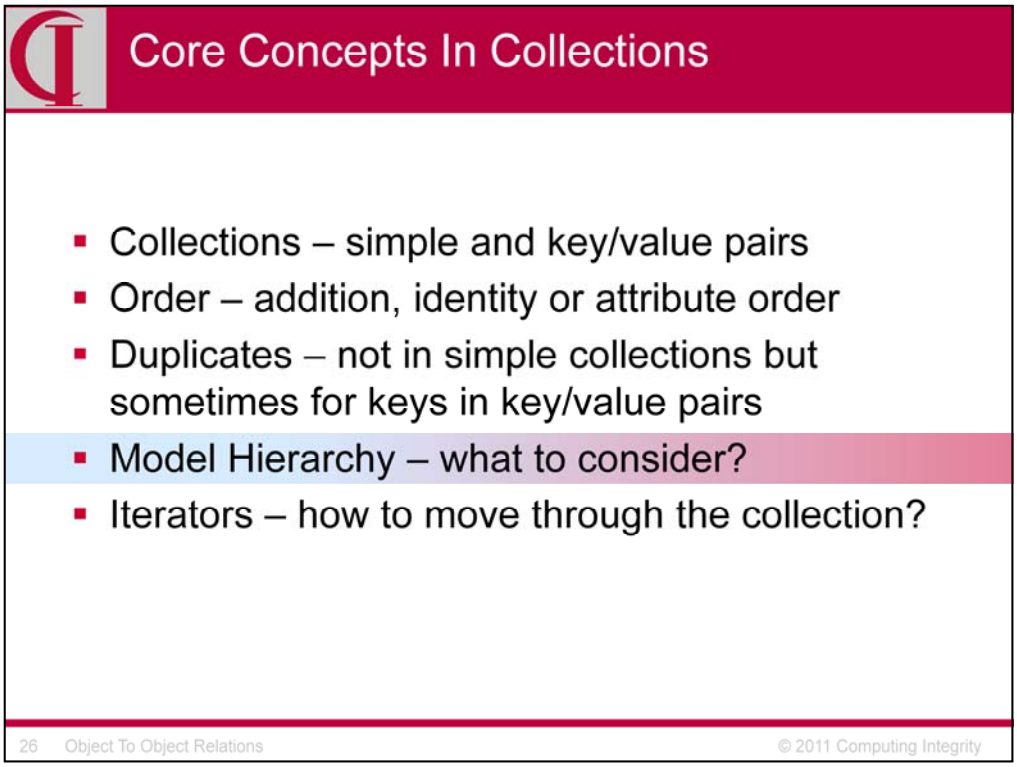


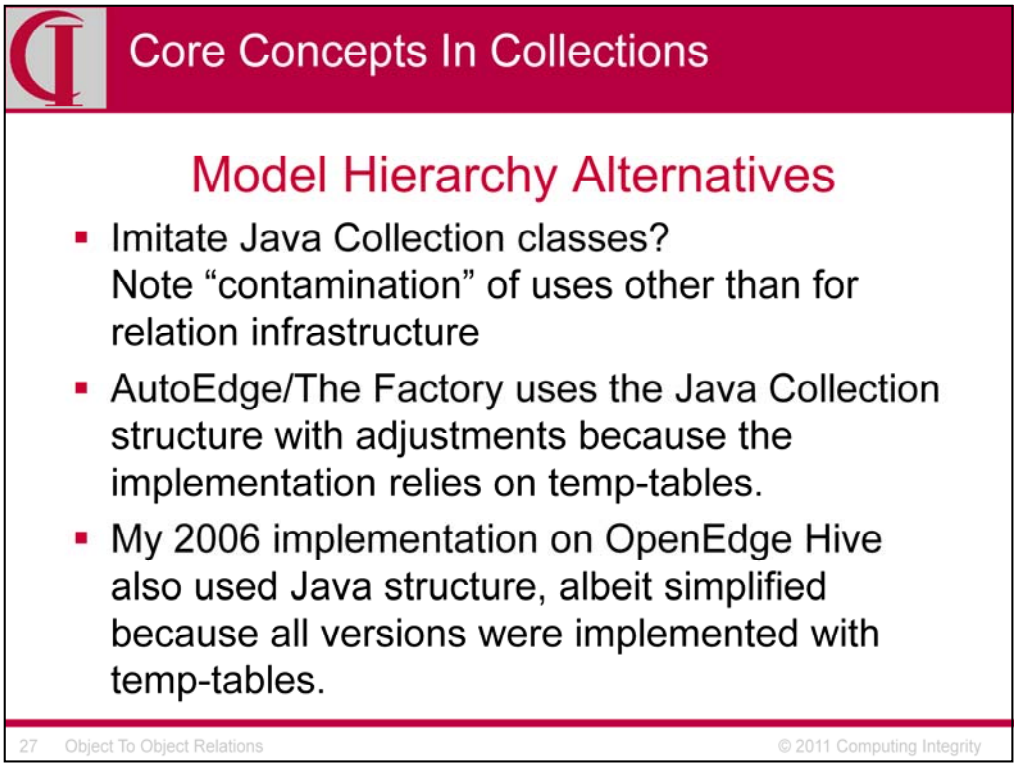
Another issue which we should consider in designing collection classes is duplicates.

In an ordinary simple collection, one would not allow duplicates because it makes no sense to have the same object in the collection more than once.

In a key/value pair collection, there is a question about whether one allows duplicate objects with different keys or no duplicate objects. If the key is an identifier, e.g., something like an order number, then we would not allow duplicates. If it is a non-unique attribute, there are cases where we might allow duplicate keys as well as cases where we might even allow duplicate objects if an object has more than one value for the same attribute.

Note that, while a mixture of unique and non-unique keys on a set of data sounds like bread and butter traditional ABL, with an OO orientation we are not thinking about all possible relationships and structures at each time but about the relationship which is relevant in the current context. Thus, typically one order and one policy about duplicates will apply to each circumstance and a simple solution for the purpose will get used in place of a very generalized solution.


Core Concepts In Collections
Collections – simple and key/value pairs
Order – addition, identity or attribute order
Duplicates – not in simple collections but sometimes for keys in key/value pairs
Model Hierarchy – what to consider?
Iterators – how to move through the collection?
26 Object To Object Relations
© 2011 Computing Integrity

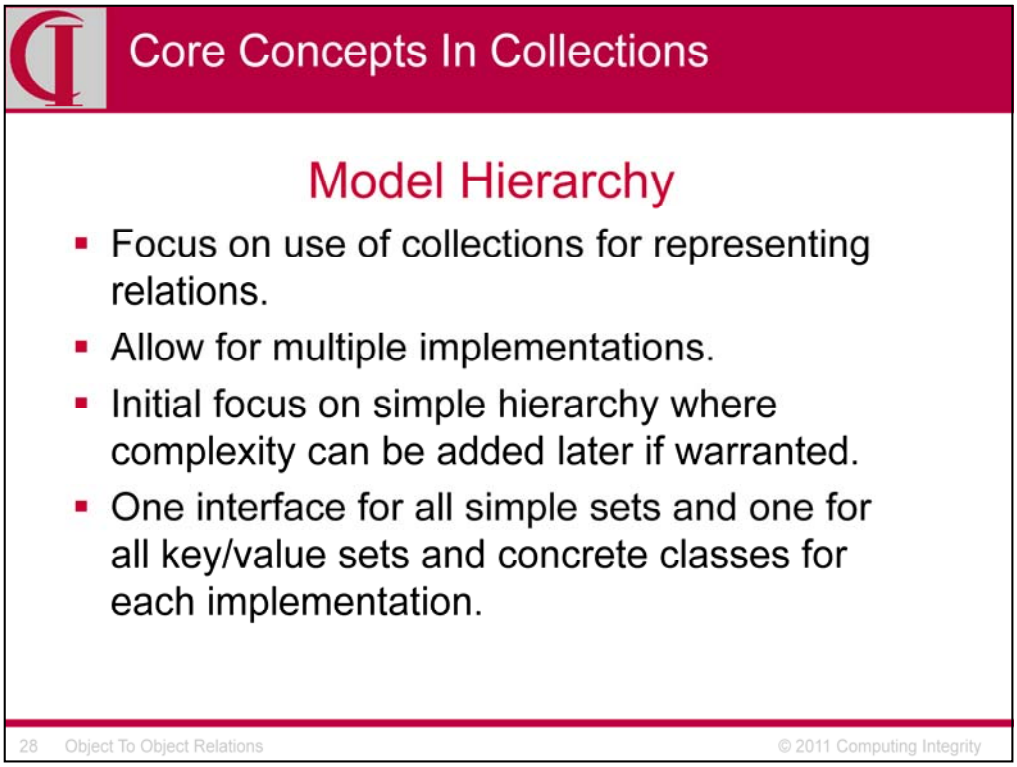


The fourth issue which we should consider in designing collection classes is the model hierarchy.

One approach is to imitate the Java Collection classes. But, this has some issues, notably “contamination” of uses other than for relation infrastructure and implementation details which are specific to Java.

AutoEdge/The Factory uses the Java Collection structure pretty faithfully with some expected adjustments because the implementation relies on temp-tables.

My 2006 implementation on OpenEdge Hive also strongly echoed an earlier version of the Java structure, albeit simplified because all versions were implemented with temp-tables.

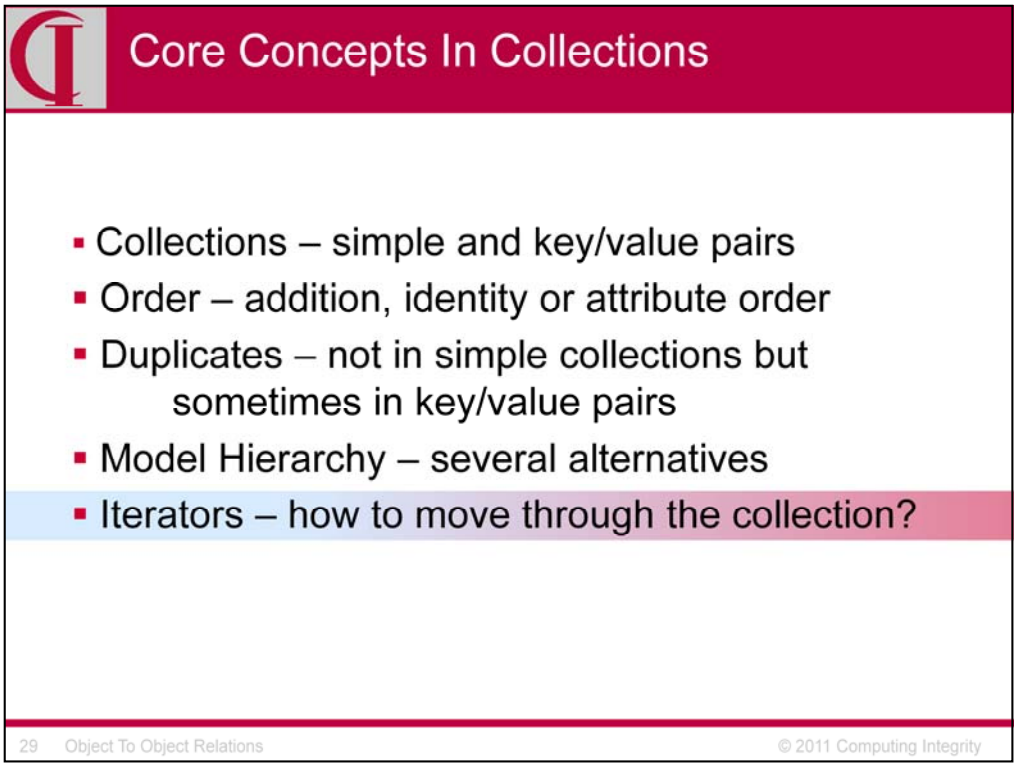


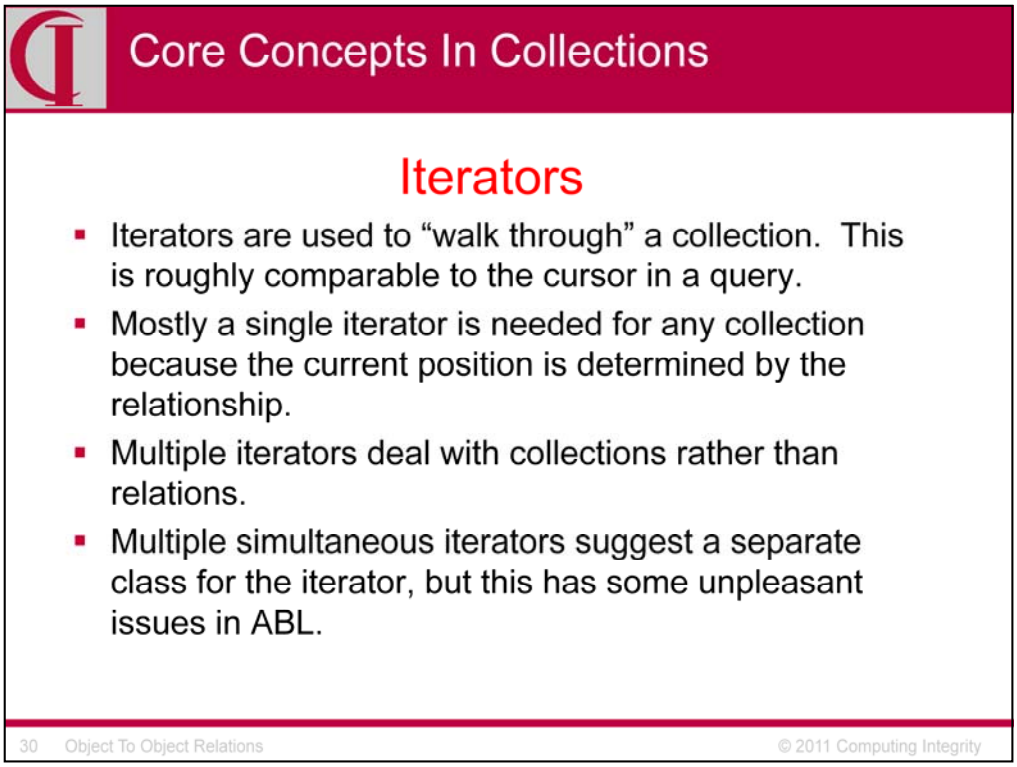
For the current discussion and implementation, we are going to focus strongly on the use of collections for representing relations and leave queues and stacks and the like for later, more specialized consideration.

However, unlike AE/TF or the 2006 effort, we want to allow for multiple implementations in the belief that different implementations will have benefits in different circumstances.

Thus, the initial focus will be on a very simple hierarchy which will only be made more complex if later development reveals the need in order to avoid code duplication.

Right now, this means one interface for all simple sets and one for all key/value sets and concrete classes for each implementation.


Core Concepts In Collections
Collections – simple and key/value pairs
Order – addition, identity or attribute order
Duplicates – not in simple collections but sometimes in key/value pairs
Model Hierarchy – several alternatives
Iterators – how to move through the collection?
29 Object To Object Relations
© 2011 Computing Integrity



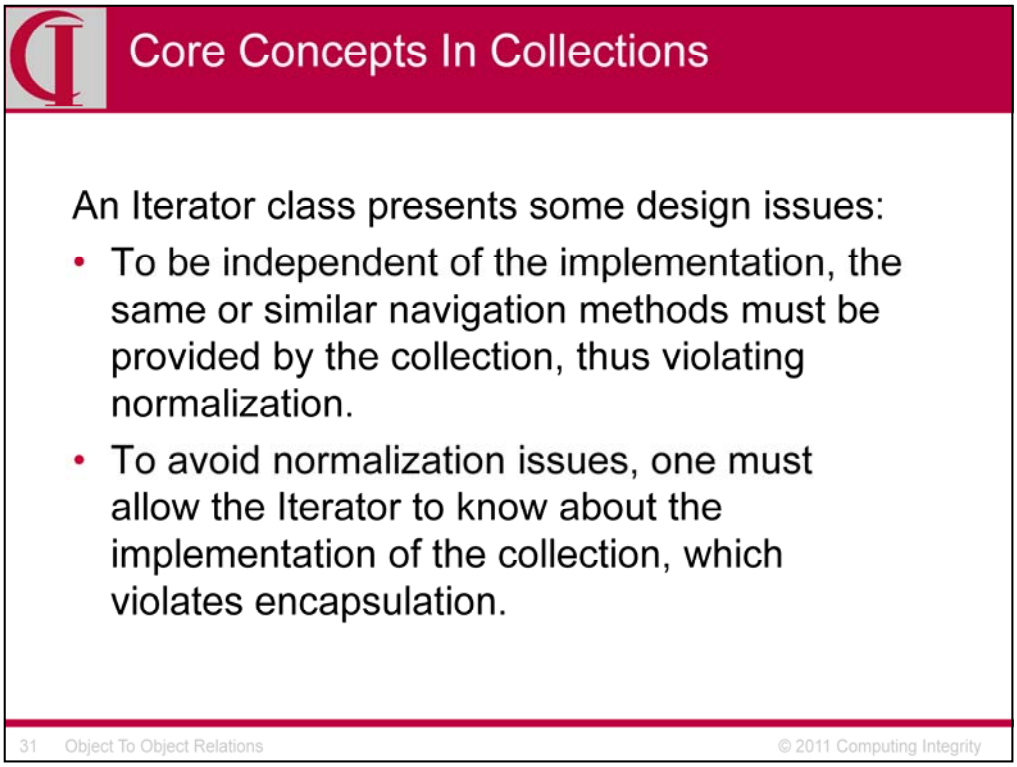
Another issue which we should consider in designing collection classes is iterators.

Iterators are used to “walk through” a collection. This is roughly comparable to the cursor in a query.

Most of the time, one needs only a single iterator for any collection because the current position is determined by the relationship.

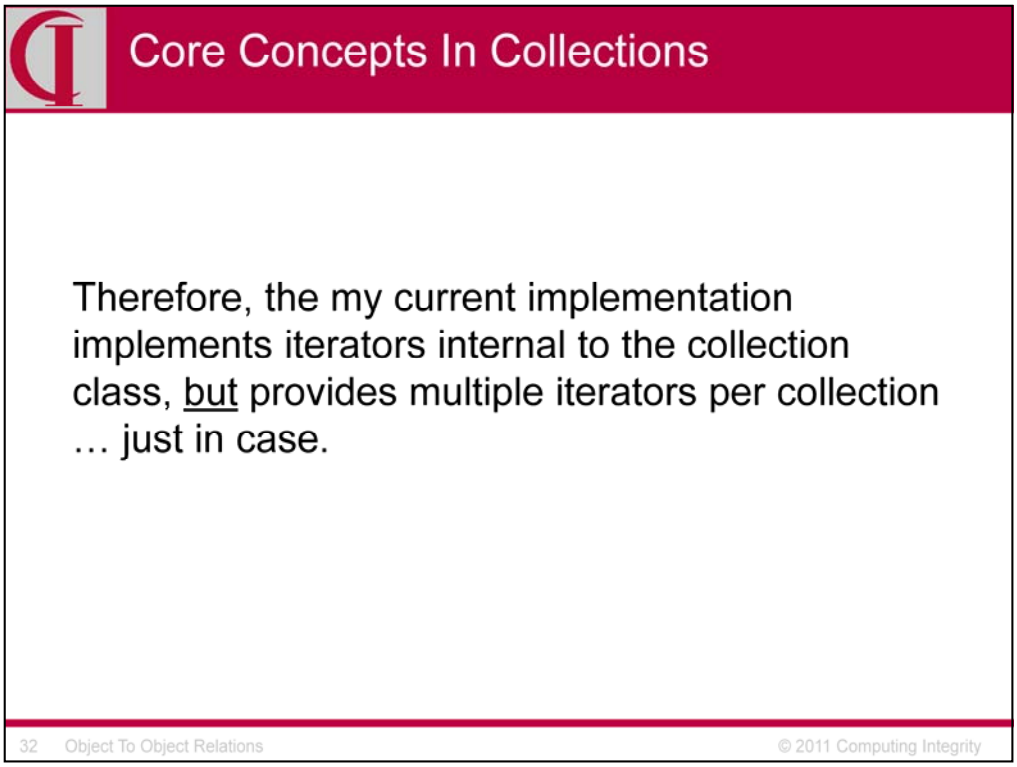
Arguments can be made for supporting multiple iterators per collection, but most of these seem to deal with collections other than ones used for a relation, i.e., perhaps they should be implemented specific to the purpose.

Multiple simultaneous iterators suggests a separate class for the iterator, but this has some unpleasant issues in ABL.



An Iterator class presents some design issues:

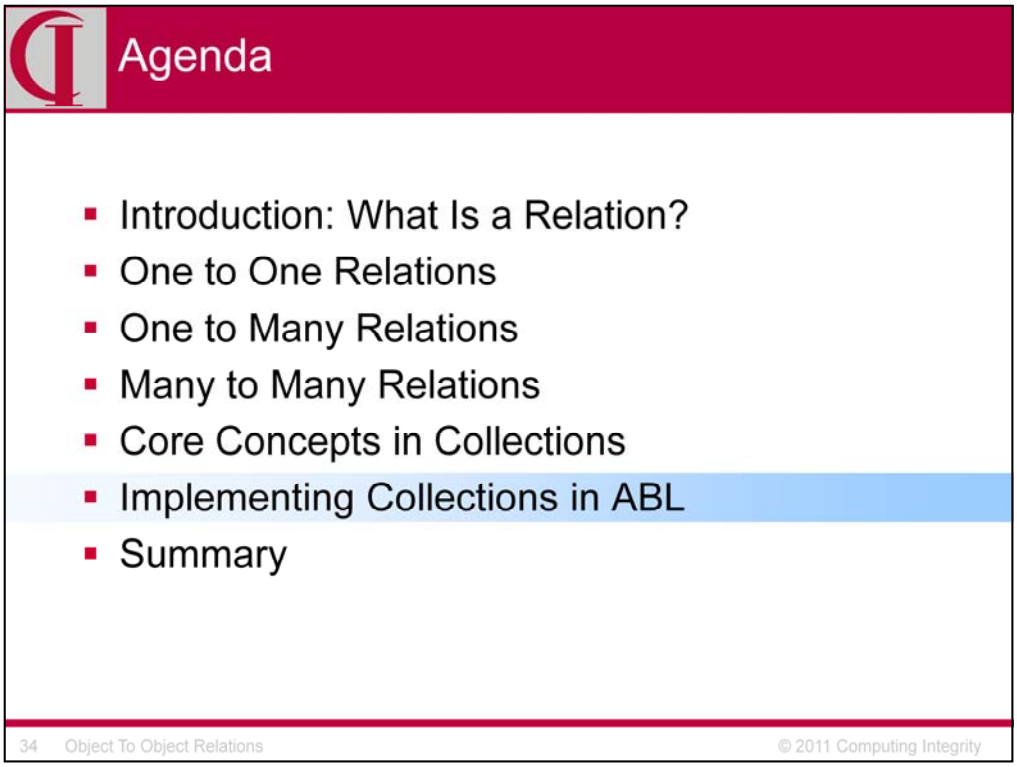
- To be independent of the implementation, the same or similar navigation methods must be provided by the collection, thus violating normalization.
- To avoid normalization issues, one must allow the Iterator to know about the implementation of the collection, which violates encapsulation.



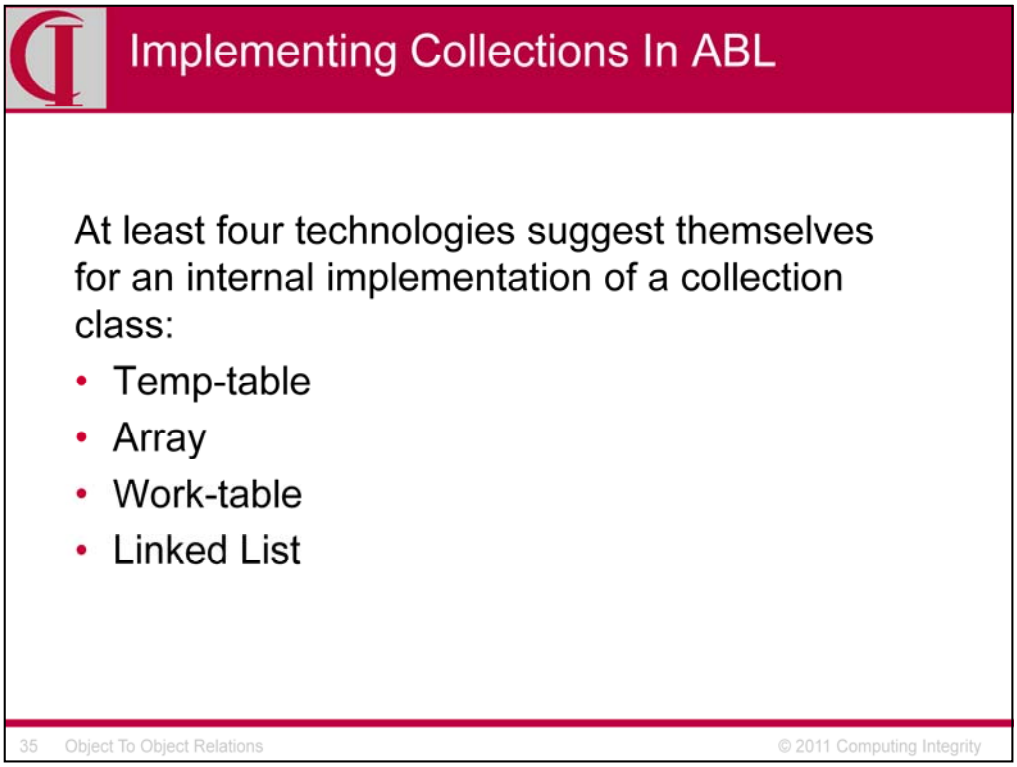
Therefore, the my current implementation implements iterators internal to the collection class, but provides multiple iterators per collection … just in case.

Core Concepts In Collections

- Collections – simple and key/value pairs
- Order – addition, identity or attribute order
- Duplicates – not in simple collections but sometimes in key/value pairs
- Model Hierarchy – several alternative
- Iterators –internal to the collection class

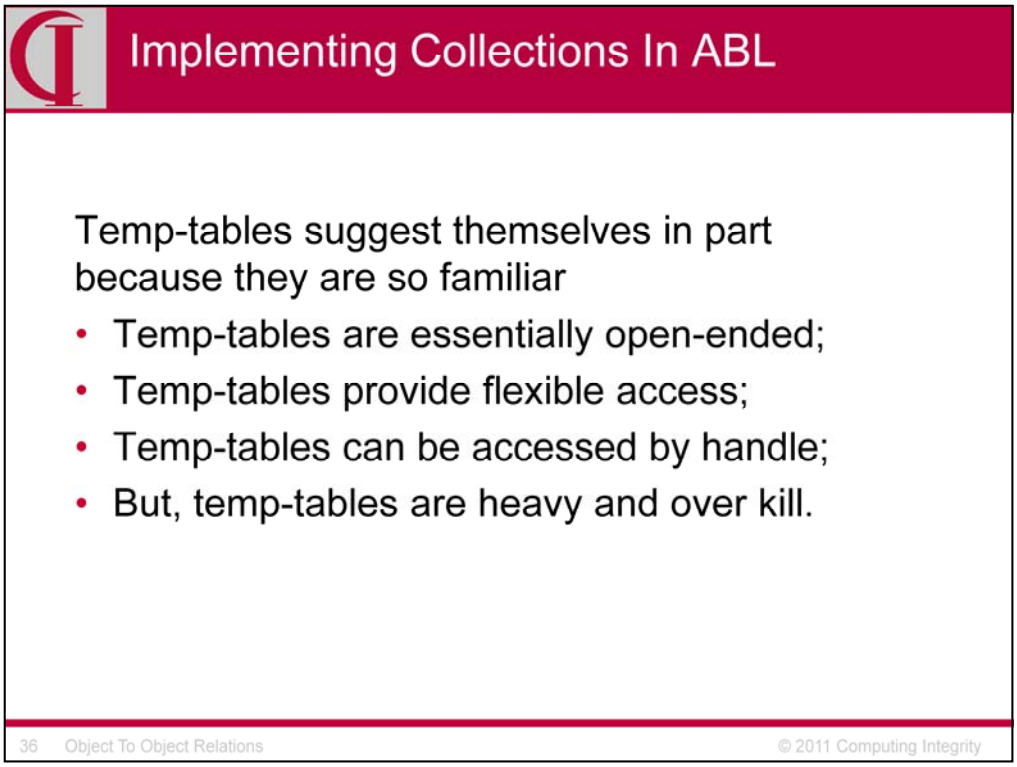


Let’s take a look at the issues involved in and the opportunities for implementing collections in ABL.



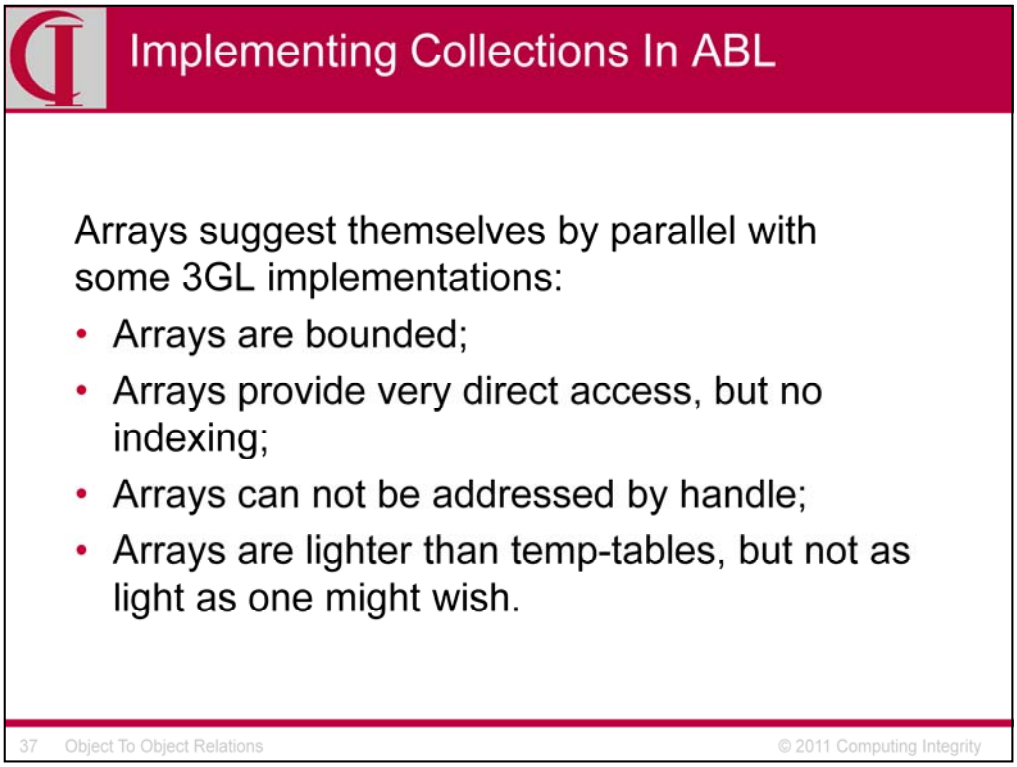
At least four technologies suggest themselves for an internal implementation of a collection class:

- Temp-table
- Array
- Work-table
- Linked List



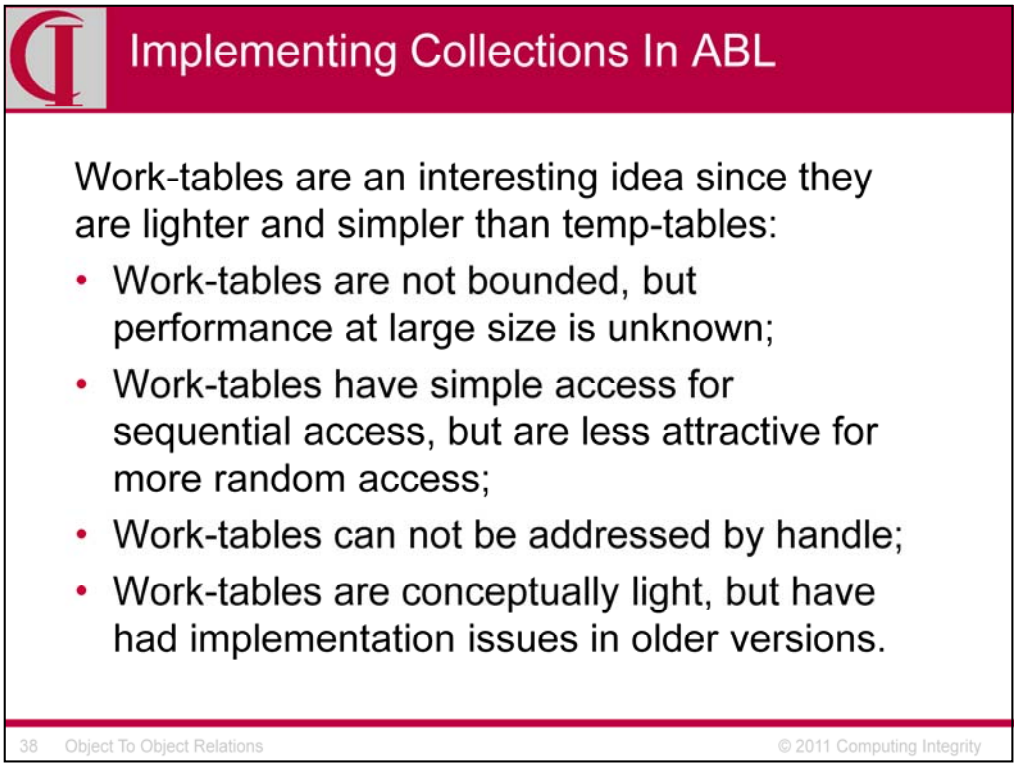
Temp-tables suggest themselves in part because they are so familiar

- Temp-tables are essentially open-ended;
- Temp-tables provide flexible access;
- Temp-tables can be accessed by handle;
- But, temp-tables are heavy and over kill.



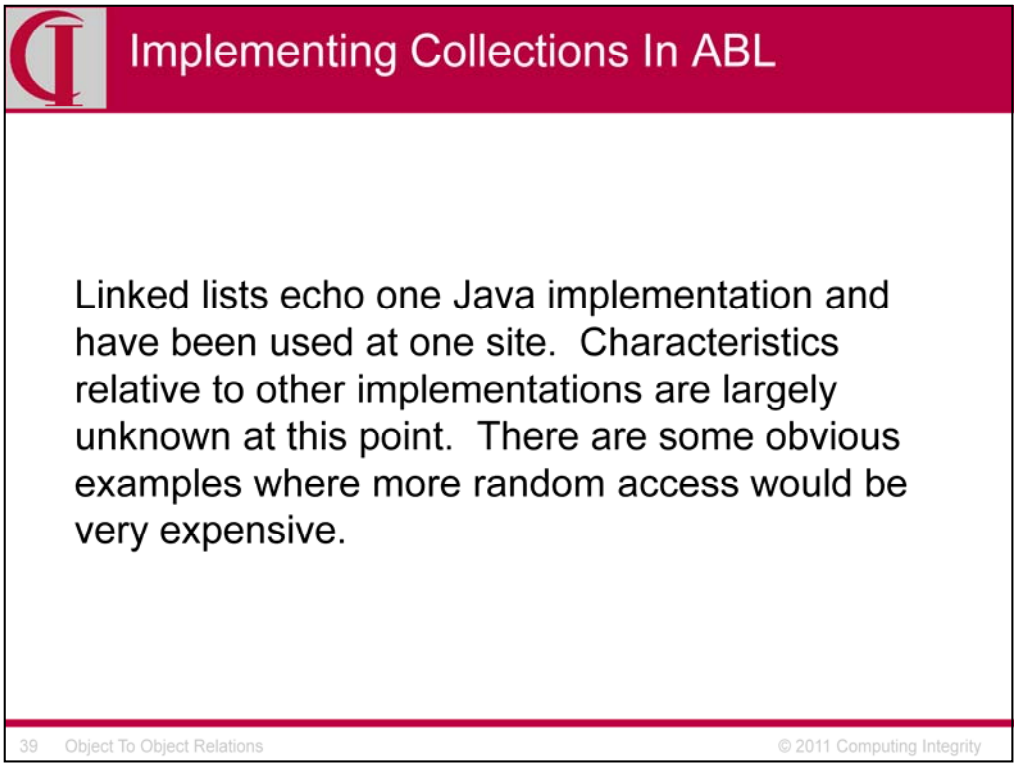
Arrays suggest themselves by parallel with some 3GL implementations:

- Arrays are bounded;
- Arrays provide very direct access, but no indexing;
- Arrays can not be addressed by handle;
- Arrays are lighter than temp-tables, but not as light as one might wish.



Work-tables are an interesting idea since they are lighter and simpler than temp-tables:

- Work-tables are not bounded, but performance at large size is unknown;
- Work-tables have simple access for sequential access, but are less attractive for more random access;
- Work-tables can not be addressed by handle;
- Work-tables are conceptually light, but have had implementation issues in older versions.

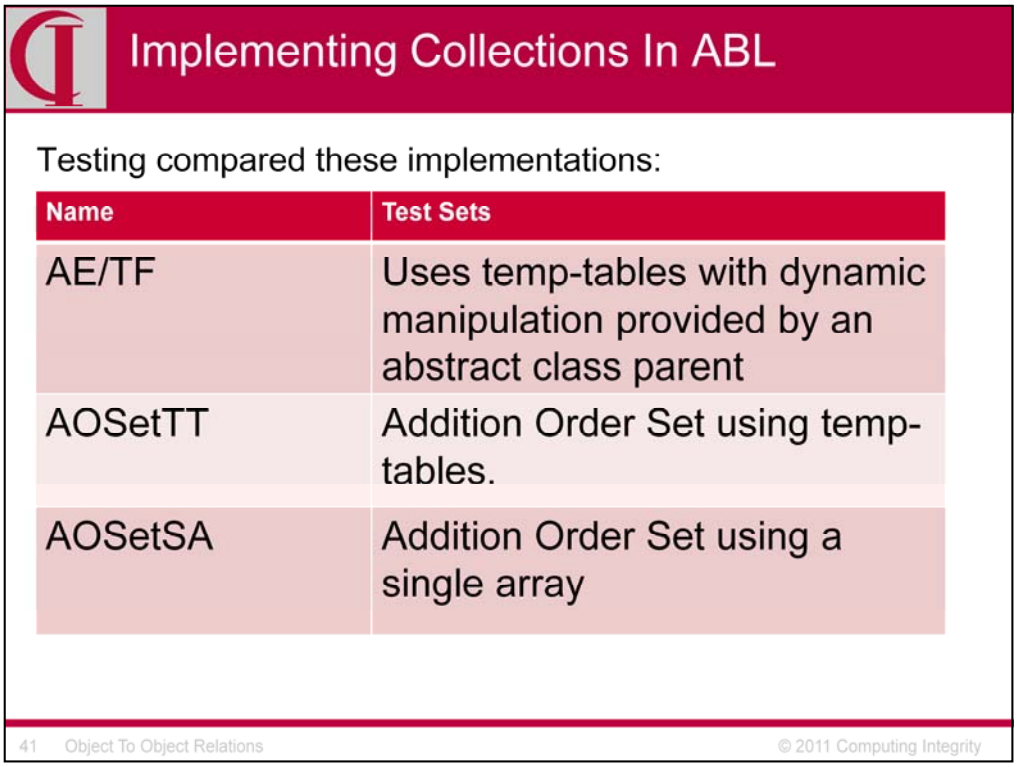


Linked lists echo one Java implementation and have been used at one site. Characteristics relative to other implementations are largely unknown at this point. There are some obvious examples where more random access would be very expensive.

Implementing Collections In ABL

Disclaimer

What follows is very much a work-in-progress. The implementation is incomplete and likely to change in response to testing and feedback.



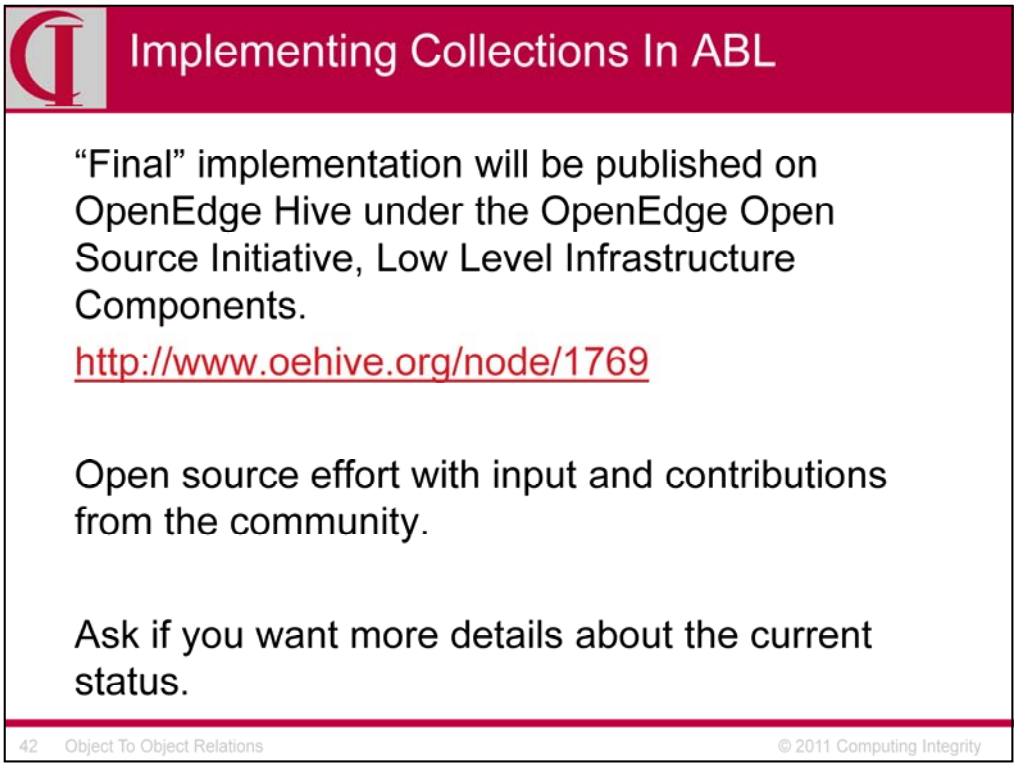
AE/TF - Uses temp-tables with dynamic manipulation provided by an abstract class parent

AOSetTT - Addition Order Set using temp-tables. Differs from the AE/TF implementation by no dynamic code

AOSetSA - Addition Order Set using a single array

Not tested, but AOSetSA compares to the previously proposed AOSetMA - Addition Order Set using multiple array implementation which Provides “natural” “growth” with limited advance knowledge of the number of items in the collection

Both AOSetTT and AOSetSA implement iSet, the interface which defines the methods of basic sets.

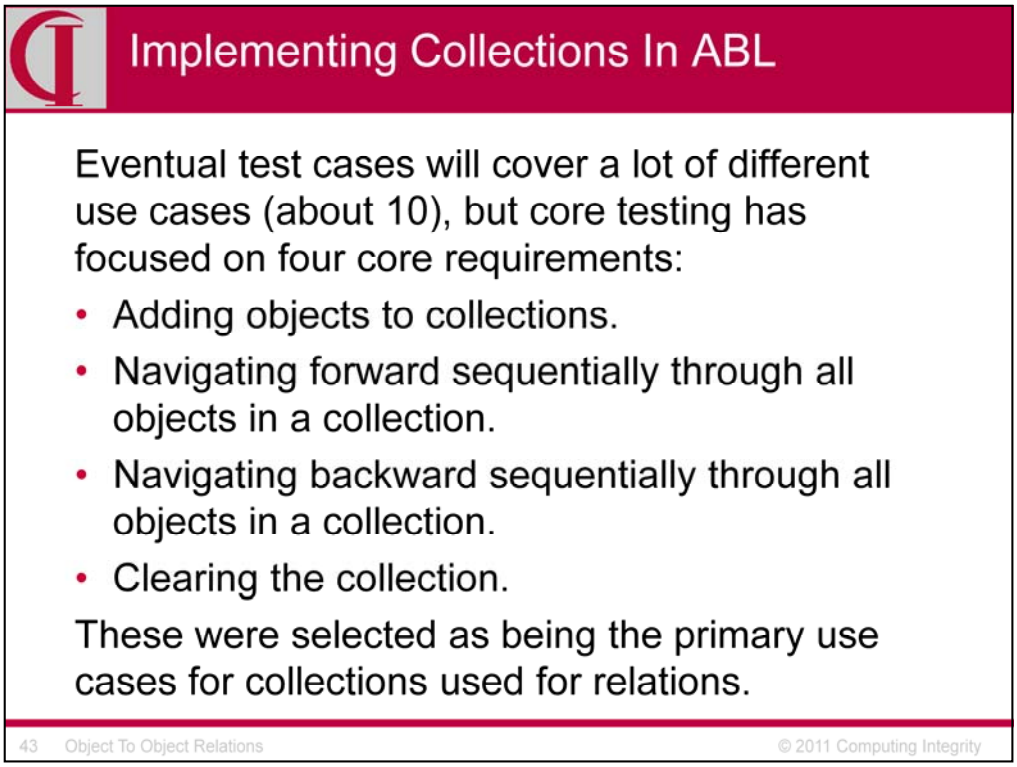


“Final” implementation will be published on OpenEdge Hive under the OpenEdge Open Source Initiative, Low Level Infrastructure Components.

http://www.oehive.org/node/1769

Open source effort with input and contributions from the community.

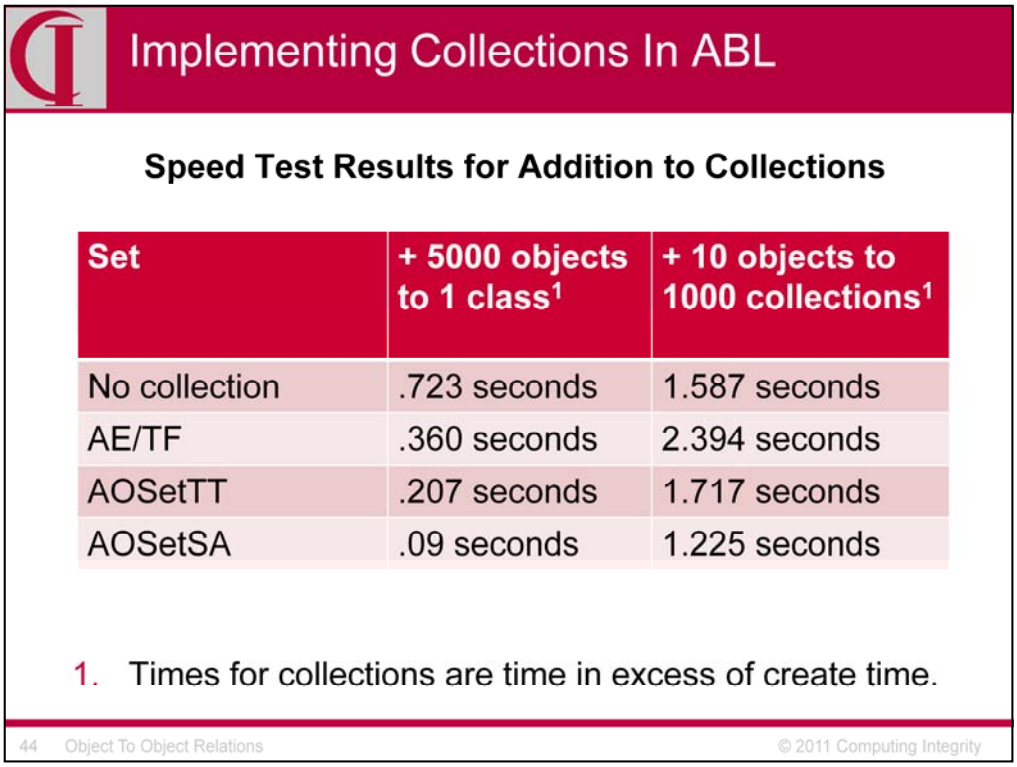
Ask if you want more details about the current status.



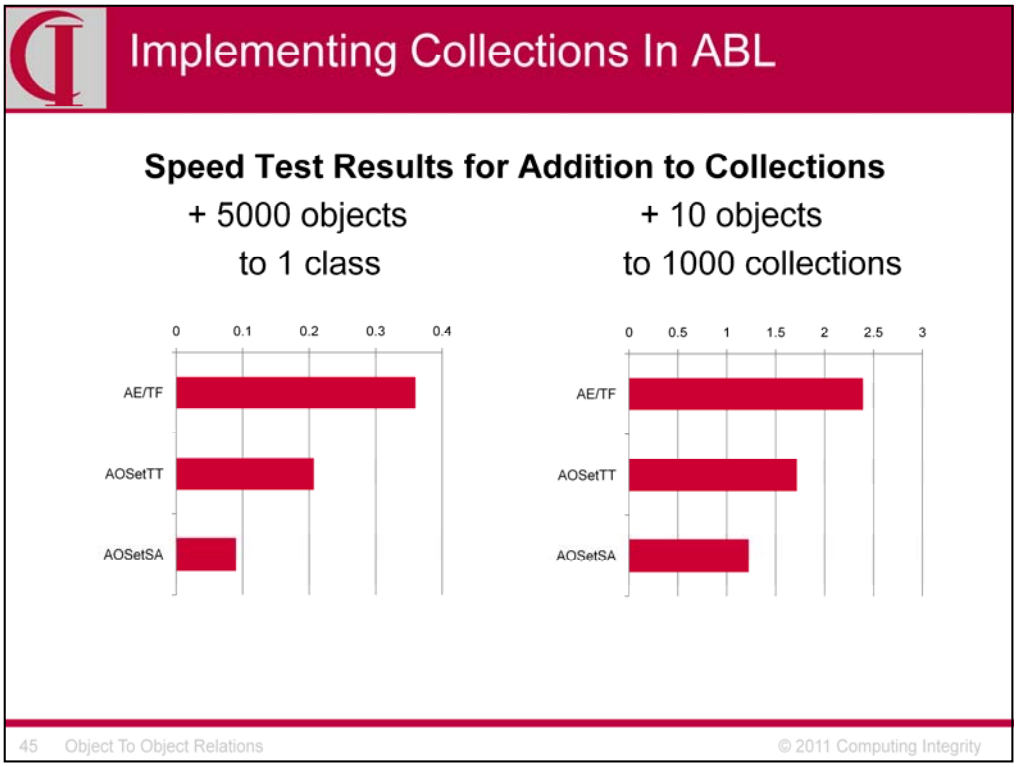
Eventual test cases will cover a lot of different use cases (about 10), but core testing has focused on four core requirements:

- Adding objects to collections.
- Navigating forward sequentially through all objects in a collection.
- Navigating backward sequentially through all objects in a collection.
- Clearing the collection.

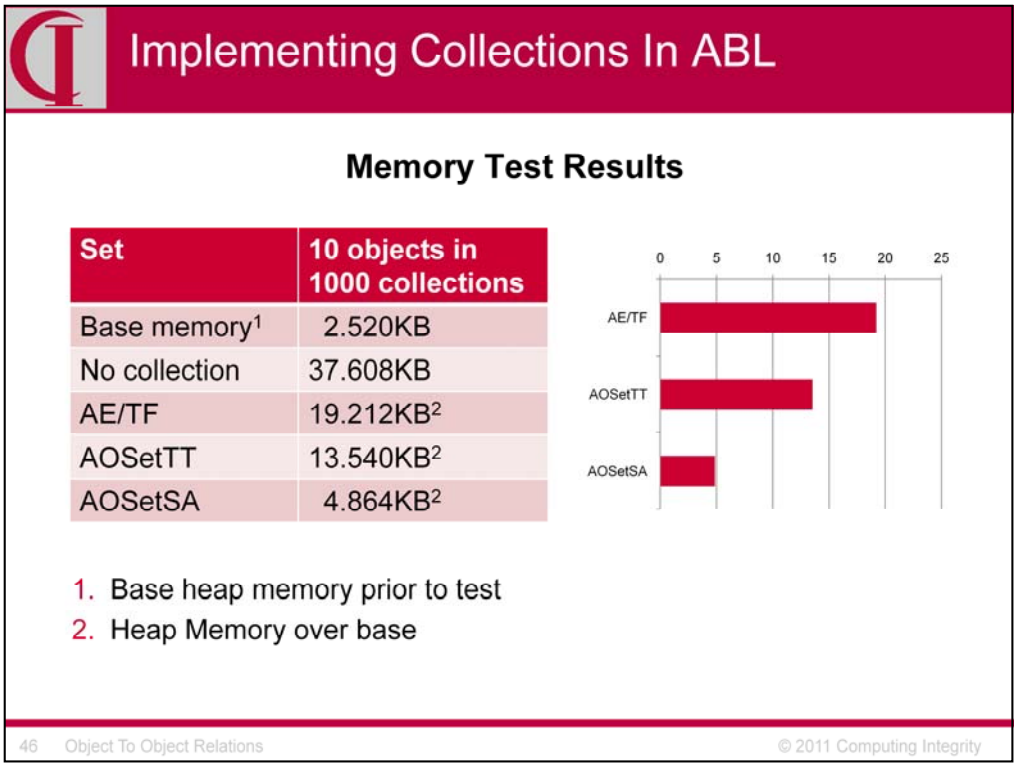
These were selected as being the primary use cases for collections used for relations.



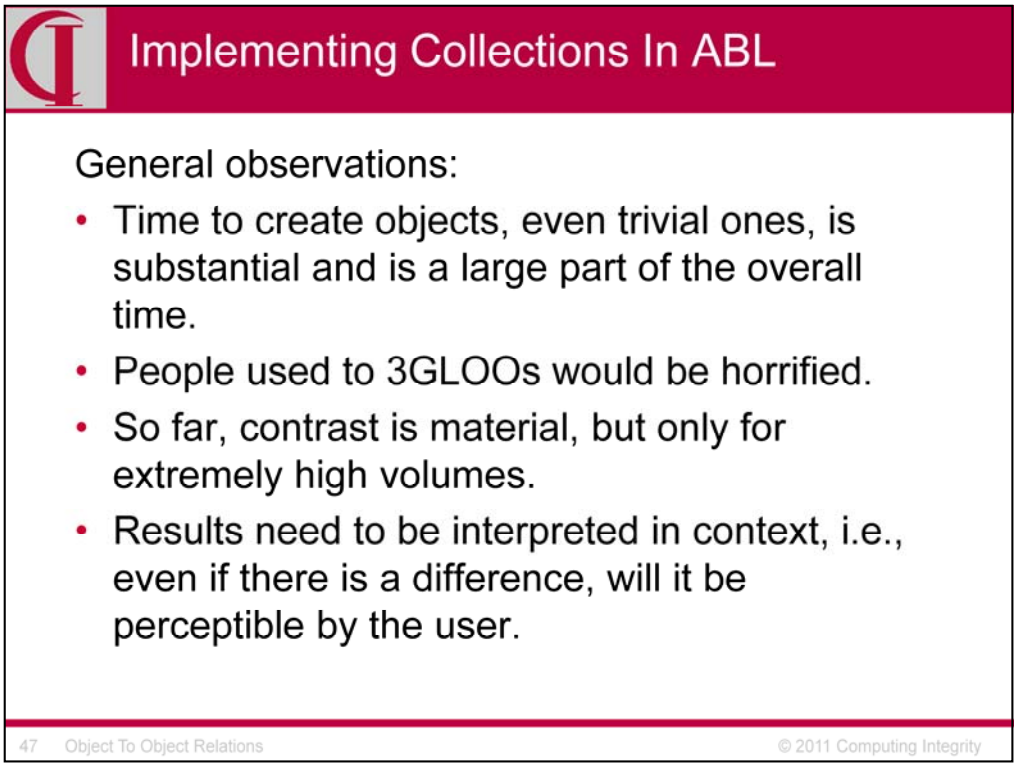
Here we see the performance test results for two tests, one adding 5000 objects to a single collection and one adding 10 objects each to 1000 collections. The time to simply create that many objects is shown on the first row. The times for each collection class type reflect the time over and above this time to create the objects. AE/TF is using the Set collection class from AutoEdge/The Factory. AOSetTT is my addition order set using a temp-table implementation. And, AOSetSA is my addition order set using a single array as an implementation.



Here is a graphical representation of these results. You can see that there are some fairly significant differences.

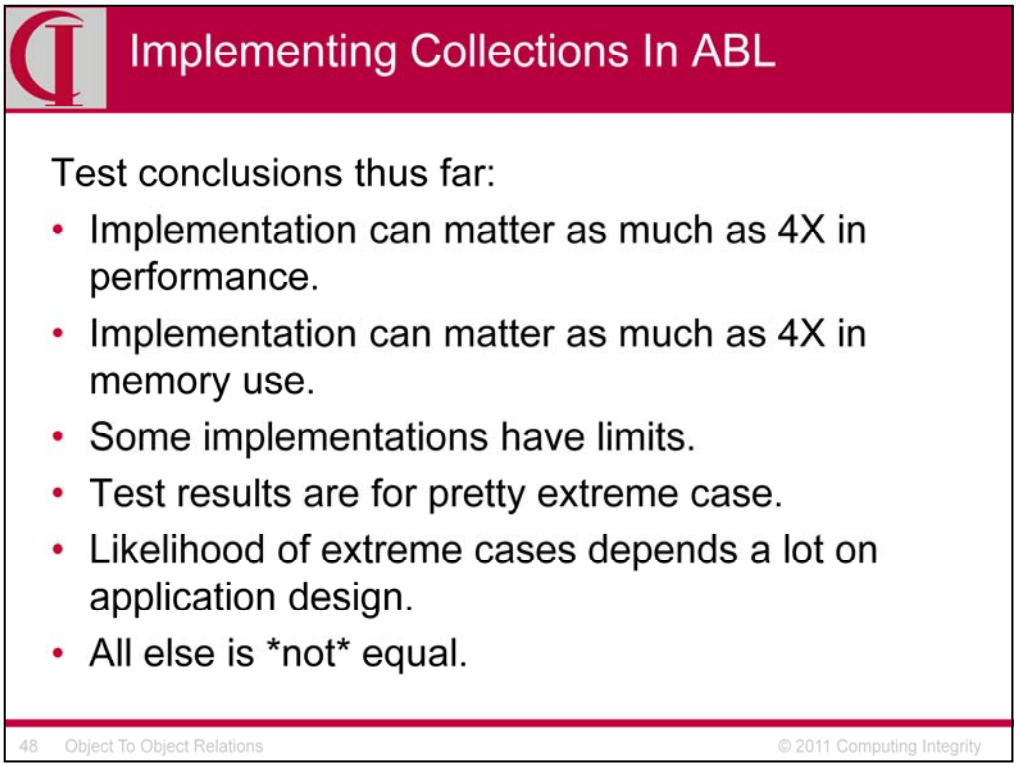


And here are the results for memory utilization. The first line is the base heap memory prior to the test. The second the memory consumed by 10000 objects not in a collection. Then there are the results for the amount of memory used by each collection class type in excess of the base needed to hold the objects alone – AE/TF for the AutoEdge implementation, AOSetTT for my temp-table implementation, and AOSetSA for my array implementation. You can see a fairly dramatic difference in memory requirements by implementation.



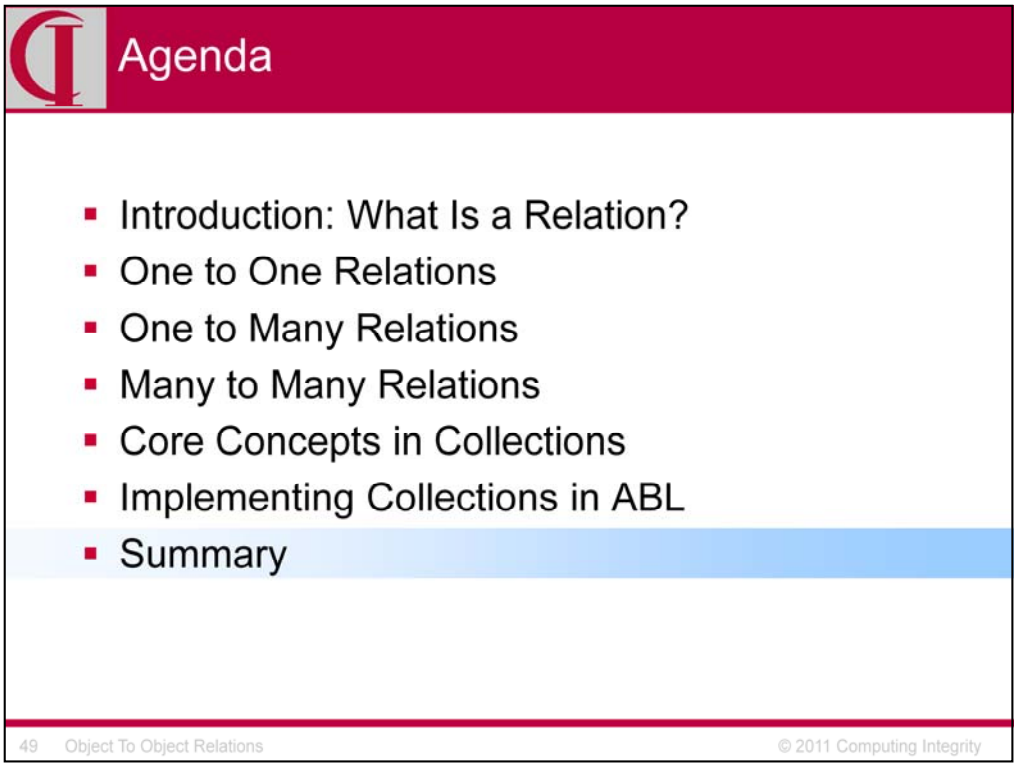
General observations:

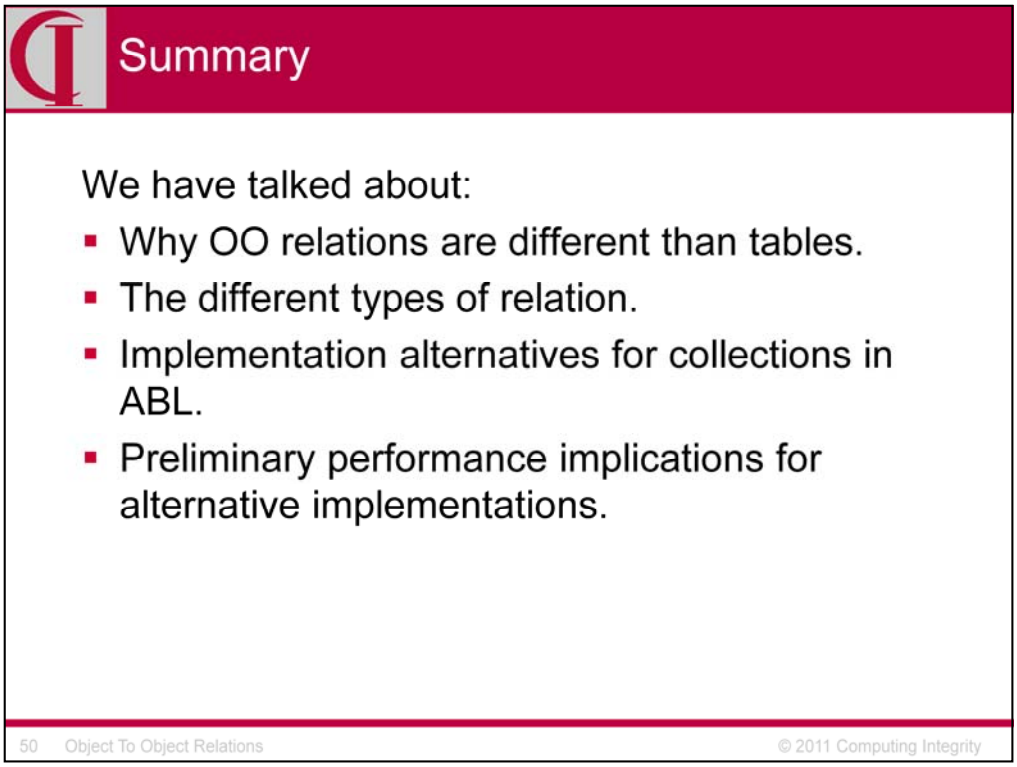
- Time to create objects, even trivial ones, is substantial and is a large part of the overall time.
- People used to 3GLOOs would be horrified.
- So far, contrast is material, but only for extremely high volumes.
- Results need to be interpreted in context, i.e., even if there is a difference, will it be perceptible by the user.



Test conclusions thus far:

- Implementation can matter as much as 4X in performance.
- Implementation can matter as much as 4X in memory use.
- Some implementations have limits.
- Test results are for pretty extreme case.
- Likelihood of extreme cases depends a lot on application design.
- All else is *not* equal.


Agenda
Introduction: What Is a Relation?
One to One Relations
One to Many Relations
Many to Many Relations
Core Concepts in Collections
Implementing Collections in ABL
Summary
49 Object To Object Relations
© 2011 Computing Integrity



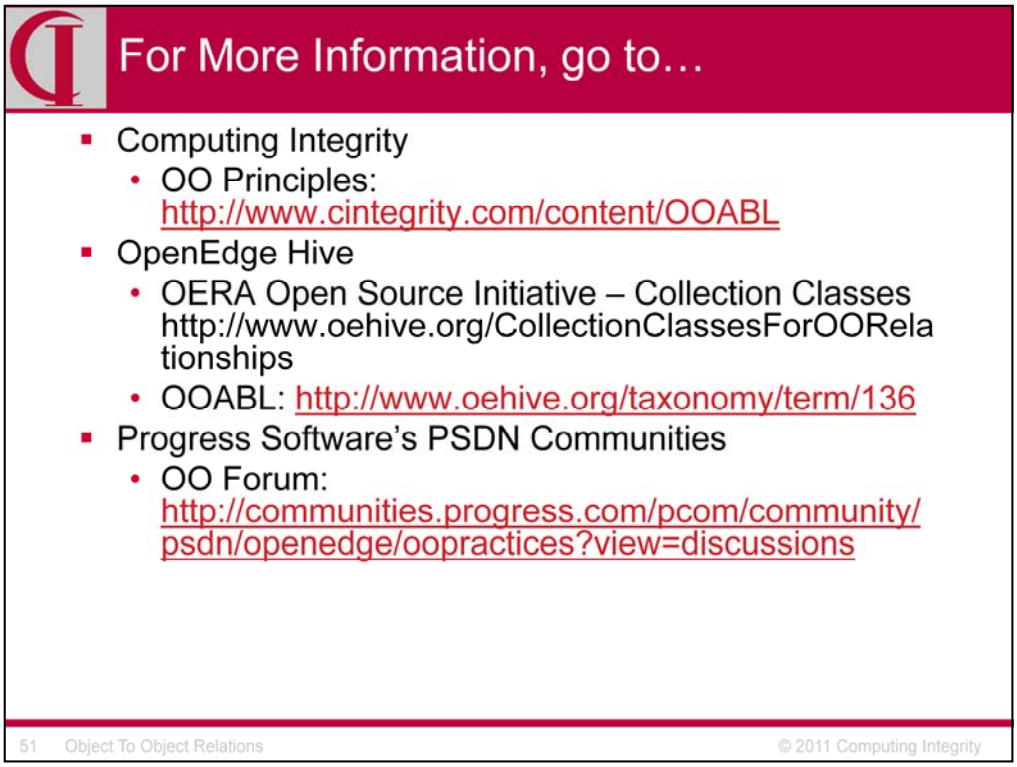
We have talked about:

Why OO relations are different than tables.

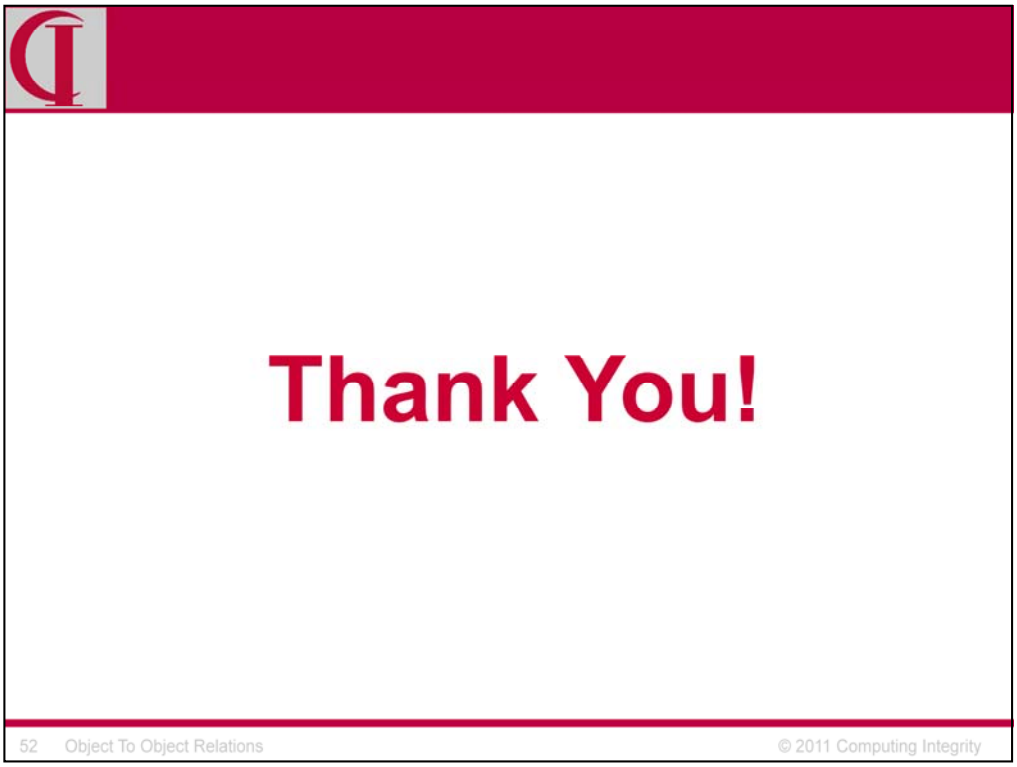
The different types of relation.

Implementation alternatives for collections in ABL.

Preliminary performance implications for alternative implementations.



Here are some links for more information. Generally, look on OE Hive under OOABL and look at the articles section of our website.


Thank You!
52 Object To Object Relations
© 2011 Computing Integrity

Thank you.

And now for questions.