# CPS122 Lecture: Representing Associations in Java; Collections

Last revised February 6, 2017

Objectives:

1. To show how associations can be represented by references
2. To introduce arrays in Java
3. To show how associations can be represented by collections

Materials:

1. Demonstration programs and projectable versions: BadReverse.java, GoodReverse.java, EvenBetterReverse.java
2. Projectable version of code excerpts illustrating summing an array, using a basic for loop and enhanced for loop
3. Projectable version of code when an association class is used

## I. Implementing Associations using Java References

A. Of course, the associations that are identified during the design phase will eventually have to be implemented in the Java code implementing the various classes. This typically takes the following form:

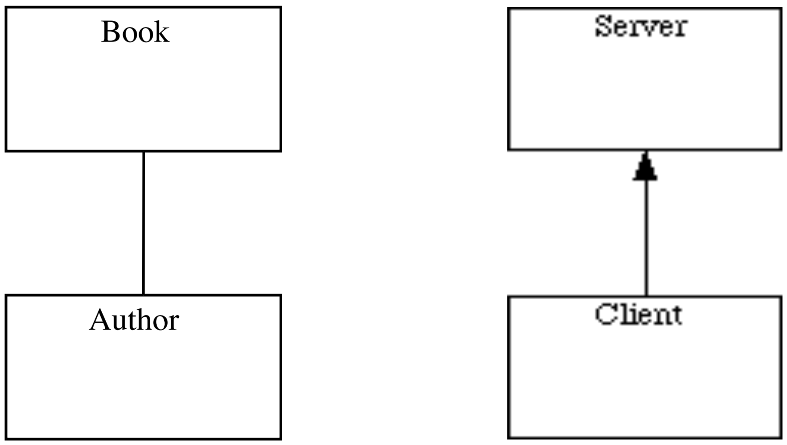
1. For each different association that relates objects of a given class to other objects, there will be a field in each object containing a link to the appropriate object(s).

    a) If the association is bidirectional, *each* participating class will need such a field.

    b) If the association is unidirectional, only the class whose objects need to know about their partner(s) will have such a field.

    *EXAMPLE:*

    Consider two cases that we looked at earlier:



In the first case, each Book object will need a field linking to the associated Author object(s), and each Author object will need a field linking to the associated Book object(s).

In the second each Client object will need a field linking to the associated Server object(s), but *not* vice-versa.

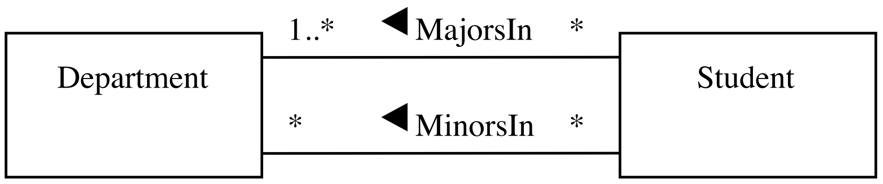
(This reduction in information that needs to be maintained is why we consider the possibility of unidirectional associations.)

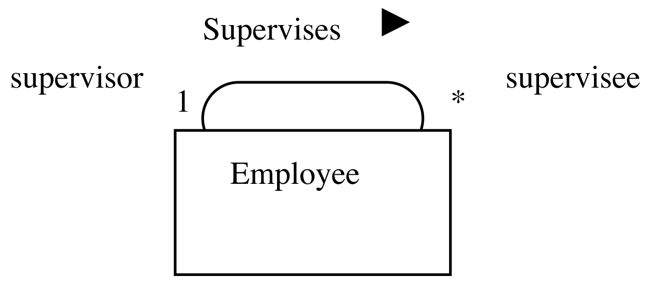
2. Frequently, the field name will be derived from the name of the association, or from the role names, if such are present. If not, the name will often come from the name of the class at the other end.

   *EXAMPLE:* A Book object may contain a field called authors, and an Author object may contain a field called books.

   *EXAMPLE:* A Client object may contain a field called server. The Server object, however, would not contain a field called client.

   *EXAMPLE:* Consider two cases we looked at earlier





In the first case, a Student object might have fields called majors and minors. A Department object might likewise have fields called majors and minors. Or the fields might be named majorsIn/ minorsIn in the Student and still probably majors and minors in the Course.

In the second case, an Employee object might have fields called supervisor and supervisees. (Note the plural in the case of the latter name - the role is supervisee, but one supervisor can supervise multiple people.) Or the fields might be named supervises and supervisedBy.

3. There are a variety of different implementation approaches that can be used to actually realize the links.

B. The simplest cases can be managed by using an ordinary variable.

1. If a given object can relate to only one other object in a given association (there is a "1" at the other end of the link), the easiest approach is to use a Java reference to the other object.

2. If the multiplicity is "0..1", the same strategy can be used, with the reference being null if there is no related object for this association.

C. More complicated cases may call for the use of an array.

1. While Python includes a number of data structures that can be used to represent collections of objects. The Java language (and, indeed, most programming languages) only directly supports one - the array. These are similar - but not at all identical to - Python lists.

2. An array is a sequence of items of some type, that can be references by position number within the array (the first item being item 0 and the last being array size - 1)

   Example: If we included the following in a program:

   ```
   Person [] persons;
   ....
   persons = new Person[10];
   ```

   then we would be able to refer to `Person[0], Person[1] ...` `Person[9]`, but not `Person[10]`.

3. If the multiplicity of an association is some fixed, small integer, or is limited by some small fixed integer, then we can use a field whose value is an array. (We will discuss arrays in Java at the end of this lecture)

   *EXAMPLE:* Suppose we assume that a given student can have at most three majors and at most two minors. Then we might include fields like the following in a Student object:

   ```
           Department [] majors;
           Department [] minors;
   ```

   And in the constructor we could include

   ```
   majors = new Department[3];
   minors = new Department[3];
   ```

   a) Of course, there are dangers if you cannot be sure that the upper limit is definite. However, three majors is probably enough for anyone!

   b) This approach is also quite wasteful of space - since we include space for storing three majors and three minors in every student object, even though most students have just one major and many have no minors!

D. When (the upper end of) the multiplicity range is “*” (or some large integer), this approach breaks down unless you know when the object is created how many other objects it will be related to (since arrays in Java are created with a fixed size). A more flexible approach results from using Collections, which we will discuss after arrays, will explore in Lab 8, and will use in later labs and the team project.

## II. Arrays in Java

A. We introduced arrays earlier as one way to represent an association.

1. Actually, arrays are used for many purposes.

2. Some Java collections (and for that matter, collections in other languages) are actually be implemented using arrays, though using them does not require any knowledge of the implementation details.

3. Most programming languages support arrays - though Python does not. The Python list is similar - but actually much more flexible, at the cost of using a more cumbersome implementation "under the hood."

B. To understand Java arrays, it is best to mentally start from scratch.

1. Recall that, in CPS121, we considered the following problem: we want to read in a list of 5 numbers and print them out in reverse order. In CPS121 we saw how Python lists could be used to solve this problem; today we will look at using Java arrays.

a) Clearly, we need to read all of the numbers before we can print any of them out. This means we have to store all the numbers in variables.

b) One solution would be to use 5 variables:

PROJECT BadReverse.java

DEMO

c) However, needing to have 5 distinct variables is cumbersome, and an approach like this would become essentially impossible if we were working, say, with 100 numbers, or 1000, or 10,000 !

d) To deal with situations like this, Java - like most programming languages - provides a built in data structure called an *array*. In Java, a variable is declared to be an array by following the type name with a pair of square brackets ([]), and individual elements in the array can be referenced by following the name of the variable with a subscript enclosed in square brackets. In particular, our example could be handled as follows:

   PROJECT GoodReverse.java

   DEMO

   Note that the complete program is now shorter than the original program - and would be much shorter if we compared programs for a larger number of values. Further, it could easily be modified to work with *any* number of numbers by changing the initial declaration of the size of the array. Every Java array has a field called length with specifies the number of elements specified when the array was created. (Note that, for arrays, this is a *field*, not a method, so no () are used.)

e) In fact, it would be easy to create a variant of this program which allows the user to specify the number of numbers when the program is run.

   PROJECT EvenBetterReverse.java

   DEMO

2. Recall that, at the start of the course, we said that Java has two basic kinds of data types: primitive types and reference types. The latter category has two subcategories - objects and arrays. Arrays in Java can be thought of as a special kind of object); however, the formal definition of the language distinguishes them because of slight differences in the way they are used. (For example, arrays have no methods).

C. To use an array in Java, you must:

1. Declare an array variable - two alternative, but equivalent syntaxes:

   `< type > [] < variable name >` (preferred)

   Example: `int [] number;`

   or

   `< type > < variable name > []` ("C" style declaration)

   Example: `int number [];`

   In this respect, Java arrays are quite different from Python lists. This requirement follows from the fact that Java is statically typed.

2. Allocate storage for the array, using new

   `< variable name > = new < type > [ < size > ]`

   a) The type used here must be the same as the type used when declaring the variable

   b) Thought the size is left unspecified when the array is *declared*, it must be specified at the time the array is *created* - it can either be an integer constant, or an integer variable or expression; in the latter case, the value of any variables at the time the array is created are what is used.)

      *Example:* `number = new int [5];`

   c) Creation can be combined with declaration

      `< type >< variable name >[] = new < type > [ <size> ]`

      *Example:* `int [] number = new int [5];`

   d) As was the case with the need for declaration, Java arrays are here quite different from Python lists.

3. Once an array has been declared and created, operations on it are quite similar to Python lists. You can:

a) Refer to the array as a whole by using its name

b) Refer to the individual elements of the array by using

`< variable name > [ < subscript > ]`

where < subscript > is an integer in the range 0 .. size - 1

Examples:
```
number[3]
number[2*i+1]
```

(*Note:* Like Python lists, Java arrays use zero-origin indexing. An array declared with size n has elements 0 .. n - 1). So, the first element is called [0], the second [1] ...

c) Note the distinction between the variable name all by itself - which stands for the *entire* array, and the variable name plus subscript, which stands for an individual *element* of the array. Operations such as arithmetic, input, and output are done on the individual elements.

Example: If a given building is a single family home, you can address mail directly to it. If it is an apartment building, you must specify a particular apartment by giving an apartment number as well. You can refer to the whole building for certain purposes - such as tax assessment - but most of the time you will need to refer to a specific apartment by number.

d) Refer to the number of elements in the array by

`< variable name > . length`

*Example:* `number.length`

D. One important characteristic of an array is that all of the elements of the array have the same type. The type of the elements of an array, however, can be any valid Java type.

1. A primitive type (boolean, char, int, etc.) - as in the example above

2. An object type. In this case, it is necessary not only to create the array, but also to create the individual elements of the array - since they are objects.

   This is typically what we will use when using an array to model an association.

3. Another array type - yielding an array of arrays, or a *multidimensional array*.

E. Array initializers

1. Ordinarily, when an array is created, its elements are initialized to the default initial value for the type involved - e.g. zero for numbers, '\000' for characters, false for booleans, or `null` for reference types.

   (`null` is a Java reserved word For any reference type, `null` is the value that means the variable does not (yet) refer to anything. It is always an error to try to execute any method of a variable that is `null`)

2. It is possible, however, to specify the initial value for an array when it is declared - in which case an abbreviated notation is used that combines declaration, creation, and initialization.

   ```
   < type >[ ] < variable name > = { < expression > , <
   expression > ... }
   ```

   EXAMPLES

   a) An array containing of all the prime integers between 1 and 20:

      ```
      int [ ] primes = { 2, 3, 5, 7, 11, 13, 17, 19 };
      ```

   b) An array of strings containing the names of the people in the first row of the room

      ```
      String [] names = { --- whatever --- };
      ```

c) Typically, when we initialize an array this way, we use *constants* as the initializers. Actually, though, it is possible to use an Java expression whose value can be calculated at the point the array is declared - but we won't pursue this further.

F. Operations on Arrays - e.g. searching, sorting etc. - are the same with Java arrays as they are with Python lists, so we'll give just one example, but otherwise we won't pursue this topic any further.

Example: Calculating the sum of all the elements in an array.

1. Suppose we have an array x of doubles. To store their sum in a variable called sum, we could proceed as follows:

```
double sum = 0.0;
for (int i = 0; i < x.length; i ++)
    sum += x[i];
```

PROJECT

This is similar to Python `for item in range ...`

2. Alternately, if we want to work with all the elements of the array - as is the case here - we could use an alternate form of the for loop (called the enhanced for loop,)

```
double sum = 0.0;
for (double item : x)
    sum += item;
```

PROJECT

(The meaning is that item is a double which should successively taken on the value of each element of the array x.

This is similar to Python `for item in list:`

## III. Collections in Java

A. As we noted earlier, while arrays can be used to model associations, there are often problems with doing so, related to the fact that the size of an array must be specified when it is created. The more common alternative is to use one of the standard collections that are part of the Java library. (The library actually implements these collections by using arrays - but this is not something we need to concern ourselves with.

B. The Collections facility is part of the `java.util` package

1. A Collection is a group of objects that supports operations like:

   a) Adding objects to the Collection.

   b) Removing an object from the Collection.

   c) Accessing individual objects in the Collection.

   (Note that an array can be thought of as a very simple and limited form of Collection, but doesn't offer the full elegance of the Collections facility in the Java library)

2. We will see in lab that Java Collections are of three basic types:

   a) *Sets* are collections in which a given object may appear at most once, and there is no ordering.

      (1) If a collection is a set, it is legitimate to ask the question "is this particular object in the collection?" (yes or no).

      (2) It is also legitimate to make the request "add this object to the collection". (Provided it's not already there.)

(3) Finally, it is legitimate to make the request "remove this object from the collection". (Provided it is in the collection to begin with.)

(4) However, it is not legitimate to ask "what is the first object in the collection"?, since sets are unordered.

(5) A simple associations is often represented using a Set.

b) *Sequences* are collections in which the contents are regarded as having some sequential order. (Note: the Java library calls these "Lists"), and multiple occurrences of an object are allowed.

(1) Sequences support all the operations we just discussed for sets.

(2) In addition, if a collection is a sequence, it is legitimate to ask questions like "what is the first object in the sequence?" or "what is the last object?" or "what is the ith object?". (Provided the collection is non-empty, in the first two cases - or has at least i+1 elements, in the last case - since elements are numbered starting at 0 - so to get, say, item 2 the collection must contain at least three elements.)

(3) It is also legitimate to make requests like "add this object at the very front" or "add this object at the very end" or "add this object in position i". (Provided the collection has at least i elements in the last case.)

(4) Finally, it is legitimate to make requests like "remove the first object" or "remove the last object" or "remove the ith object". (Provided the collection is non-empty, in the first two cases - or has at least i+1 elements, in the last case.)

(5) Sequences are commonly used for associations that have an inherent order (denoted by the UML stereotype `{ ordered }`)

c) *Maps* are collections of key-value pairs. (Actually, Java Maps are not technically Collections due to some implementation issues, but it is common to speak of them as "small c" collections.

   (1) Though the same value may occur many times in the map, each key is associated with at most one value.

   (2) If a collection is a map, it is legitimate to ask questions like "what value - if any - is associated with the following key?" or "does this map contain the following key?".

   (3) It is also legitimate to make the request "put the following key-value pair in the map". This can have one of two effects:

      (a) If the key was not in the map to begin with, it is added with the specified value.

      (b) If it was in the map, but with a different value, the old value is removed and the new value is associated with the key.

   (4) Finally, it is legitimate to make the request "remove the following key from the map". If the key was in the map, it and its associated value are removed; if not, nothing happens.

   (5) Maps are often used for qualified associations, with the qualifier serving as the key, and the associated object the value

      EXAMPLE: The qualified association between a college and its students could be represented by a map (stored in the college object) consisting of pairs where the student id is the key and the corresponding student object is the value.

d) For all types of Collections, it is possible to create an Iterator object that makes it possible to access each item in the collection once.

(1) For sequences, the order in which items are accessed by an Iterator is the sequential order first, second ...

(2) For sets and maps, the order is implementation-determined.

(3) In the case of maps, the Iterator is actually obtained from either its set of keys or its set of values. (Most often from its set of keys)

3. The Java Collections library contains two or more implementations for each of the different types of collection. The different implementations of a given type of Collection have the same behavior, but have different performance characteristics.

   a) For List (Sequence), the Java library supplies:

      (1) LinkedList - good if the list changes size (grows or shrinks) frequently), good for accessing either end of the list, but slower when accessing items in the middle of the list

      (2) ArrayList - good if accessing elements by specific position, but slower for adds and removes.

   b) For Set, the Java library supplies:

      (1) HashSet - more efficient in most cases

      (2) TreeSet - an iterator will access the elements of the set in a specific order based on their value (e.g. Strings would be kept in alphabetical order.)

   c) For Map, the Java library supplies:

      (1) HashMap - more efficient in most cases

      (2) TreeMap - an iterator obtained from the key set will access the elements of the map in key order.

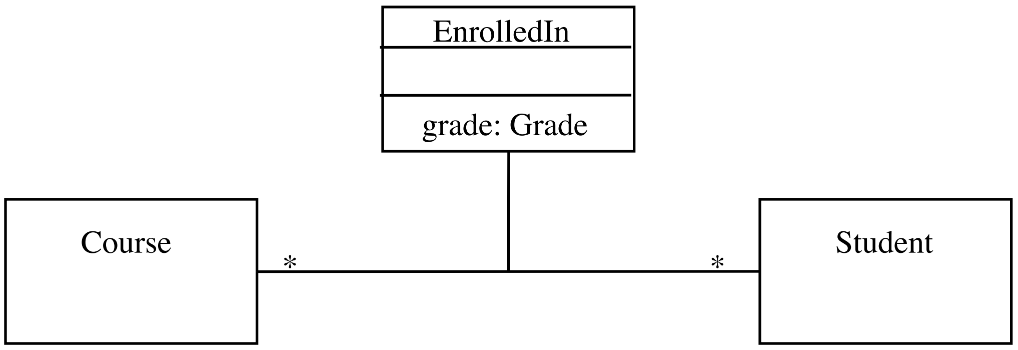
4. In an upcoming lab, one of your major tasks will be to identify appropriate ways to represent various associations using the different types of collection. (Just to make it interesting, you will need to use several different types :-))

C. We noted earlier that if an association has attributes associated with the association itself (not just the participating objects), an association class can be used. In this case:

1. Each participating object contains a reference to the association class object.

2. The association class object contains references to each of the related objects.

*EXAMPLE:*



a)

```
class Course {
    (Some sort of collection of references to
      EnrolledIn objects) enrolledIn;
    ...
  }

class Student {
    (Some sort of collection of references to
      EnrolledIn objects) enrolledIn;
    ...
  }
```

```
class EnrolledIn {
    Course course;
    Student student;
    Grade grade;
    ...
  }
```

PROJECT

D. Now, let's think about the various associations in the Library problem and how they might be represented by Java collections: For simplicity, we will make the simplifying assumption that the library collection only includes books. It will also prove helpful to assume that there is a singleton object (perhaps of a class called LibraryDatabase) which represents the database and "owns" all the other objects.

1. First though, we need to consider an interesting (and actually quite tricky) issue that arises in connection with rentable items.

    a) Typically, a library carries multiple copies of popular books.

        (1) Presumably, we need a separate object for each copy, since each can be rented to a different patron, be due on a different day, etc..

        (2) At the same time, we want to associate all kinds of information with a copy - its call number, title, author - but storing all this information multiple times (once for each copy) is problematic

            (a) Wasteful of space

            (b) Makes extra work when a new copy comes in

            (c) Suppose we needed to correct a piece of information fir some reason. We would need to make this change in <u>each</u> copy.

(3) Moreover, when we accept a reservation for a particular item, we don't want to associate the reservation with a <u>specific</u> copy - we want to associate it with the item itself. Why?

ASK

b) There is a very problematic way to handle a case like this:

(1) Create a separate <u>class</u> for each item. Thus, we would have a class for each book we carry.

(a) The various items of information we would want to record about an item could be represented as <u>static</u> fields of the appropriate class - e.g. we would have something like

```
class StudentGuideToOODevelopment {
  static String callNumber = "QA76.9.O35";
  static String title = "A Student Guide to
    Object-Oriented Development";
  static String author = "Britton, Carol";
     ...
```

(b) A copy would be represented by an object of the appropriate class. Thus, if we had 3 copies of the book, each would be represented by an object of class `StudentGuideToOODevelopment` - 3 in all.

(2) Why is this very problematic?

ASK

(a) A serious problem is that we have to create a new class every time a new book is acquired. This means that the software would have to be revised and recompiled frequently!

(b) A similar problem is that any change to information stored about a book would necessitate revising and recompiling the software.

c) A much better way is to make use of a <u>design pattern</u>, which is, in essence, a good solution to a commonly-occurring tricky problem. (We will talk about design patterns later in the course.) In this case, we want to use a pattern known as the Abstraction-Occurrence pattern.

(1) In essence, what we want to do is to use <u>two</u> classes to represent items.

(a) One class - which we might call something like `Item` - represents the abstraction.

i) There would be, then <u>one</u> object of class `Item` for A Student Guide to OO Development, another for `Java Backpack Guide` ...

ii) This one object holds all the information that pertains to all copies of the item - call number, title, author.

iii) When a new item is added to our collection we create a new object of this class to represent it.

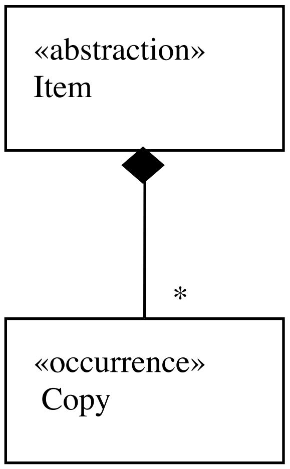
(b) A second class - which we might call `Copy` - represents the occurrence.

i) There would be one object of this class for each physical copy of an item that we own - thus, if we had 3 copies of A Student Guide ..., there would be 3 objects of class `Copy` representing them.

ii) Each object of class `Copy` would be associated with the object of class `Item` to which it pertains.

(c) If the library owned 10,000 copies in all, representing 5000 different items, there would, in total, be 10,000 objects of class `Copy`, and 5000 objects of class `Item`.

(d) We might show this in UML as follows:



i) Composition is appropriate here, because,

(1) At least as far as the library is concerned, an item is composed of its copies

(2) Each copy is associated with one and only one item

(3) A copy of one item can never be associated with a different item

ii) A multiplicity of * is appropriate on the `Copy` end, because a given `Item` can have an arbitrary number of copies. (We might even have 0 for a short time, if our only copy is lost and we're waiting for a reorder to come in)

iii) Bidirectional navigability is appropriate, because me want to be able to answer both of the following kinds of question:

(1) What copies of "A Student Guide to OO Development" do we own?

(2) What is the author of the item that a QA76.9.O35 copy 1 is a copy of?

2. Of course, we need to have an association between the library and the items it owns. Clearly this is 1..*.

   a) What navigability is appropriate here?

      ASK

   b) What type of collection is appropriate in the LibraryDatabase object to hold references to Item objects for this case?

      *ASK*

      Map, keyed on something like the call number of the item. The LibraryDatabase object would have a declaration like

      `Map <String, Item> items;`

      But nothing would be needed in the Item object, since we have no need to navigate from a Item to the (one and only) LibraryDatabase.

3. Again, we need an association between the LibraryDatabase and the copies it owns as well. Clearly, this is also 1..* When a patron wants to checkout a copy, the call number and copy number on the copy is used to find the appropriate object to associate with the patron.

   a) What navigability is appropriate here?

      ASK

   b) What type of collection is appropriate in the `LibraryDatabase` object to hold references to `Copy` objects for this case?

      *ASK*

`Map`, keyed on the copy's call number and copy number (which could be concatenated to form a single String). The `LibraryDatabase` object would have a declaration like

```
Map <String, Copy > copies;
```

But nothing would be needed in the copy object, since we have no need to navigate from an item to the (one and only) `LibraryDatabase`

4. The association between the library and its patrons is clearly 1:*. When a patron wants to check out one or more copies, his or her phone number is used to access the stored information on the patron.

   a) What navigability is appropriate here?

   ASK

   b) What type of collection is appropriate in the `LibraryDatabase` object to hold references to `Patron` objects for this case?

   *ASK*

   `Map`, keyed on the patron's phone number. The LibraryDatabase object would have a declaration like

   ```
   Map < String, Patron > patrons;
   ```

   But nothing would be needed in the `Patron` object, since we have no need to navigate from a Patron to the (one and only) `LibraryDatabase.`

5. We have already said that we need an association between a patron and the specific copies the patron has out.

   a) What is the multiplicity here?

   0..1 : * - a copy is either out to 1 patron, or it's on the shelf; but a patron can have any number of copies out

b) In this case, we need bidirectional navigability, so we can answer questions like

(1) What copies does patron 555-1234 have out?

(2) What patron rented copy `QA76.9.035 copy 1` (that was just returned)

c) Observe that the act of checking out an item involves attributes - e.g. date due. We can represent this either by using an association class object, or by recording the date due as an attribute of the copy (since it can only be checked out to one person at a time)

ASK class for how each approach would be set up in terms of Java collections

6. We also need an association between a patron and the items he/she has reservations for.

a) What is the multiplicity here?

ASK

It must be * .. * - any number of patrons can have a reservation for a given item, and a patron can have reservations for multiple items.

b) In this case, we probably want bidirectional navigability, so we can answer questions like

(1) Who has reservations for `QA76.9.035`?

(2) What reservations does patron 555-1234 have?

c) Presumably, we want to keep track of reservations on a first-come first-served basis. This being the case, what sort of collection is needed in an Item object to keep track of the patrons who have a reservation for the item?

ASK

7. What would we need to do to this structure to also handle dvd?

   ASK

   a) Make Item an abstract base class with concrete subclasses Book and DVD.

   b) Copies and reservations are still associated with an Item. But certain operations (e.g. determining a due date) are delegated to the appropriate concrete class via abstract methods. That's all we need to do!