Chapter 7

Relationships between Objects

An object-oriented program consists of a set of objects that collaborate to achieve the objectives of the program. As a result, very few objects in an object-oriented program stand alone. Most of them collaborate with other objects in a number of ways. For an object to collaborate with another object, it must be related to that object in some way. Thus, it is important to understand how objects relate to each other and how to implement these relationships in an object-oriented program.

A relationship is a connection among objects. The three most important relationships in an object-oriented program are *dependencies*, *associations*, and *generalizations*. In a UML diagram, a relationship is typically drawn as a line between two objects. Different kinds of lines are used to distinguish the kinds of relationships possible. This chapter describes the three main types of relationships in an object-oriented program and explains how to implement each type of relationship in Java.

7.1 Dependencies

A class `A` *depends* on a class `B` if objects of class `A` manipulate objects of class `B` in any way. For example, a `Mailbox` class in a voice mail system depends on a `Message` class because the `Mailbox` class manipulates `Message` objects (Horstmann, 2004). A *dependency* is essentially a *using* relationship that states that a change in specification of the class being used (e.g., the `Message` class) may affect the other class that uses it (e.g., the `Mailbox` class). However, the converse is not necessarily true.

It is sometimes helpful to consider when a class does not depend on another class. If a class can carry out all of its tasks without being aware that some other class exists, then it does not depend on that class. Also, a dependency relationship between two classes `A` and `B` implies that `A` is coupled to `B`. Thus, a

dependency relationship between the `Mailbox` class and the `Message` class indicates that the `Mailbox` class is coupled to the `Message` class. Consequently, the goal of low coupling can be achieved by minimizing the number of dependency relationships in an object-oriented program.

In a UML diagram, a dependency relationship is drawn as a dashed directed line, with the arrow pointing to the class being depended on. Figure 7.1 shows the dependency relationship between a `Mailbox` class and a `Message` class in a voice mail system (Horstmann, 2004).

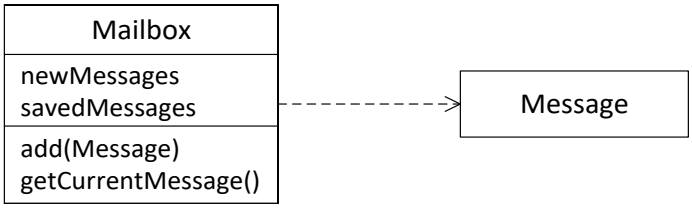

Figure 7.1: Dependency between `Mailbox` and `Message`

7.2 Associations

An *association* is a structural relationship between two classes that specifies that objects of one class are connected to objects of the other class. Given an association connecting two classes, it is possible to navigate from an object of one class to an object of the other class, and vice versa (Booch, Rumbaugh, and Jacobson, 1999). An association that connects exactly two classes is called a *binary association*. The classes involved in a binary association are usually distinct; however, it can sometimes be the same. Associations that connect more than two classes are called *n-ary associations* where *n* is the number of classes involved in the association; however, these are not as common as binary associations.

In the UML, a binary association is drawn as a solid line connecting the same or two different classes. Consider the association between a `Person` class and a `Company` class (Booch, Rumbaugh, and Jacobson, 1999). This association is shown in Figure 7.2.

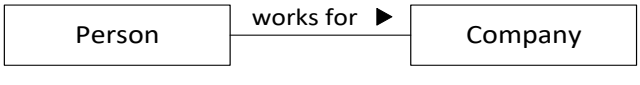

Figure 7.2: Association between a `Person` class and a `Company` Class

The association is given a name, `works for`, which is written just above the association line. To remove ambiguity, a direction reading arrow is used to indicate the direction in which the association name should be read. So, the association in Figure 7.2 can be expressed as, "*Person works for Company*".

7.2.1 Multiplicity of an Association

Since an association is a structural relationship between objects, it is important to state how many objects may be connected across an instance of an association. This value is called the *multiplicity* of an association and is written as an expression that evaluates to a range of values or an explicit value. The UML symbols used for common multiplicity values are shown in Table 7.1:

Multiplicity Value	*Symbol Used*
Exactly one	1
Zero or one	0..1
Many	0..*, *
One or more	1..*
An exact number	e.g., 5

Table 7.1: Symbols Used for Multiplicity Values

Figure 7.3 shows three associations where the multiplicity of the association from one object to the other is (a) one-to-one, (b) one-to-many, and (c) many-to-many, respectively.

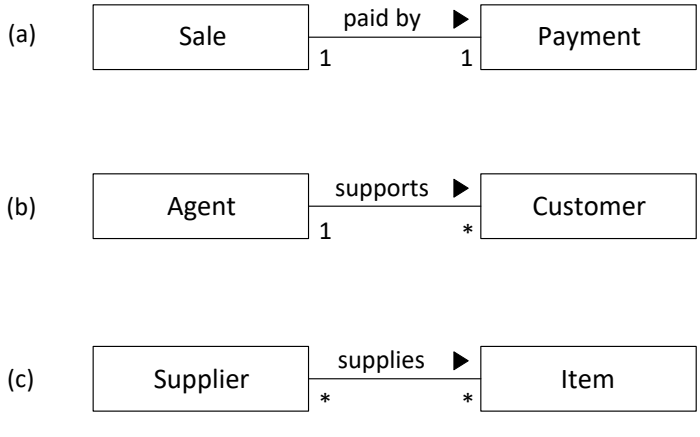

Figure 7.3: Associations with Different Multiplicity Values

Figure 7.3(a) indicates that a `Sale` is paid for by one `Payment` in a point-of-sale application (Larman, 1998). Figure 7.3(b) says that one `Agent` supports many `Customer`s in an insurance application. Figure 7.3(c) says that one `Supplier` can supply many `Item`s and that one `Item` can be supplied by many `Supplier`s in an inventory application (Date, 1995).

7.2.2 Aggregation

An *aggregation* relationship models a "*whole/part*" relationship between two classes `A` and `B`, in which an instance of `A` (the "whole") consists of instances of `B` (the "parts"). Aggregation represents a "*has-a*" relationship, meaning that an object of the whole *has* objects of the part. Aggregation is really just a special kind of association. It is drawn using an association line with an open diamond at the "whole" end. Figure 7.4 shows how to draw the aggregation relationship between a `Company` and its `Department`s (Booch, Rumbaugh, and Jacobson, 1999).

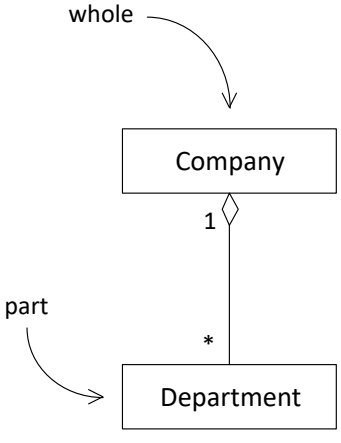

Figure 7.4: Aggregation Relationship between `Company` and `Department`

7.2.3 Composition

Composition is a form of aggregation with strong ownership between the "whole" and its "parts". In a composite aggregation, the "whole" is responsible for the creation and destruction of the "parts". The "parts" in the association live and die with the "whole". The "parts" can belong to only one composite at any point in time.

In the UML, composition is drawn exactly like aggregation except that the diamond at the "whole" end is shaded. Figure 7.5 shows a composition

relationship between a `Customer` and his/her `Account`s at a financial institution. A `Customer` object can be associated with many `Account` objects over a period of time. However, the lifetime of the related `Account` objects is dependent on the lifetime of the `Customer`. So, if a `Customer` object is removed, the `Account` objects corresponding to that `Customer` must also be removed.

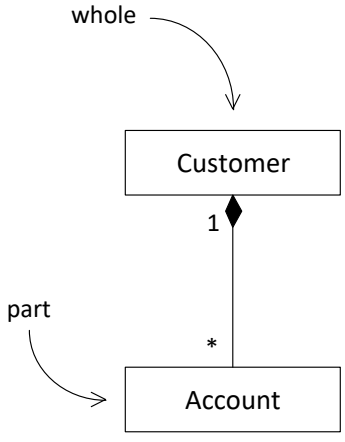

Figure 7.5: Composition Relationship between a `Customer` and `Account`

7.3 Generalizations

A *generalization* is a relationship between a general class (called the *superclass* or *parent class*) and a more specific class (called the *subclass* or *child class*). Generalization relationships are often called "*is-a*" or "*is-a-kind-of*" relationship since the child class *is-a-kind-of* the parent class. In a generalization relationship, the child class inherits the state and behavior of its parent. So, at the least, an instance of the child class can act like an instance of the parent.

As an example, consider the generalization relationship between a `Person` class and a `Customer` class. The `Customer` class inherits the state and behavior of its parent class `Person`. However, it may also add state and behavior of its own. The relationship is shown in Figure 7.6.

Observe that the line for the generalization relationship has a hollow arrow at one end pointing to the parent class, `Person`. The other end of the line is connected to the child class, `Customer`.

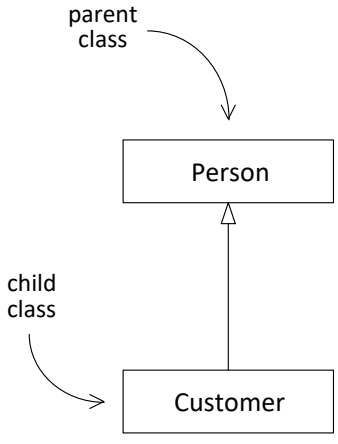

Figure 7.6: Generalization Relationship between `Customer` and `Person`

Generalizations will not be discussed further in this chapter. This is an important topic in object-oriented programming and it is covered in detail in Chapter 9.

7.4 Implementing Dependency Relationships

If instances of a class `A` is dependent on (or uses) instances of a class `B`, it follows that `B` provides a service (through a method) that is required by instances of class `A`. This service can be requested by invoking the appropriate method on an instance of class `B`, and sending the appropriate arguments.

Suppose that `a` is an instance of `A`, and that `b` is an instance of `B`. Suppose further, that `a` has access to `b`. If `a` wishes to request a service, `s()` of `b`, this can be done as follows:

```
b.s();
```

In general, the above statement can be used to request any service `s()` from an instance `b` of any class `B`, except that it will have to be modified when arguments are required. For example, suppose that `s()` requires an integer argument. Then, `s()` can be requested as follows:

```
b.s(10);              // argument is an integer value
```

There are several ways in which `a` can get access to `b`:

- `a` can create an instance of `B` and assign it to `b`

- `b` can be passed as a parameter to a method `m()` of the class `A` (the parameter must be of type `B`)
- `b` can be returned by a method `m()` of another class `C` that is called by `a`. `m()` can be an instance method or a class method.

Examples of these three ways in which `a` can get access to an instance of `B` are shown in Table 7.2 below. As would be expected, these also correspond to three common ways for `A` to be coupled to `B`.

How to get access to an instance of B	*Sample code*
An instance of `B` is created by `A`	B b; b = new B();

An instance of `B` is a parameter of a method of `A`	public void m (B b) {
	  // body of method
	}

An instance of `B` is returned by a method `m()` of a class `C` (an instance of `C` is required)	B b;
	C c; // assign c to an
	  // instance of C
	b = c.m();

An instance of `B` is returned by a class method `m()` of a class `C`	B b; b = C.m();

Table 7.2: Getting Access to an Instance of Class `B` from Class `A`

7.5 Implementing Associations

This section explains how to implement binary associations. Implementing a binary association depends on the multiplicity of the relationship between the participating objects; thus, the common multiplicities are discussed separately. The section also explains how an association class can be used to implement a binary association of any multiplicity.

7.5.1 Implementing One-to-One Associations

Consider the one-to-one association between a `Sale` object and a `Payment` object shown in Figure 7.3. A `Sale` represents the event of a purchase transaction and a `Payment` represents payment for the items that make up the `Sale`.

Suppose that we are given a `Sale` object and we wish to find the corresponding Payment object. All that is needed is to store a reference to the `Payment` object in the `Sale` object. However, given a `Payment` object, it will not be possible to find the corresponding `Sale` object. If bi-directional

navigability is required of the `paid by` relationship (i.e., the ability to navigate the relationship from both ends), a reference to the `Sale` object must also be stored in the `Payment` object. This makes it possible to traverse the `paid by` relationship from either a `Sale` object or a `Payment` object.

The object references that are required to implement bi-directional navigability are illustrated in Figure 7.7. An object reference is denoted as a small circle with an arrow that points to the actual object in memory. Note that accessor and mutator methods may now have to be written to view and modify the object references in `Sale` and `Payment` (giving due consideration to design guidelines such as the Law of Demeter).

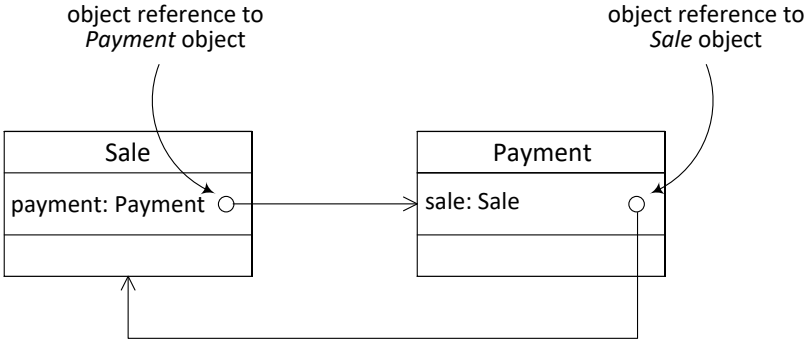

Figure 7.7: Implementation of the One-to-One `paid by` Association

7.5.2 Implementing One-to-Many Associations

Consider the one-to-many association between an insurance `Agent` class and a `Customer` class based on the `supports` relationship shown in Figure 7.3. The one-to-many association from `Agent` to `Customer` can be implemented as a collection of references to `Customer` objects in `Agent` (Rumbaugh et al. 1991). The simplest collection to use is an array of `Customer` objects. The `Agent` class will now have to provide methods to add and remove `Customer` objects from the array as well as methods to query the `Customer` objects in the array.

If, given a `Customer`, we wish to find out who is the `Agent` who supports that `Customer`, an object reference to an `Agent` object must be stored in `Customer`. Implementation of the object references for the `supports` relationship is shown in Figure 7.8. In the diagram, there are three `Customer` objects currently being supported by the given `Agent` object. The collection of `Customer` objects in `Agent` therefore contains three references to `Customer` objects. Each `Customer` object contains a single reference to the `Agent` they are supported by.

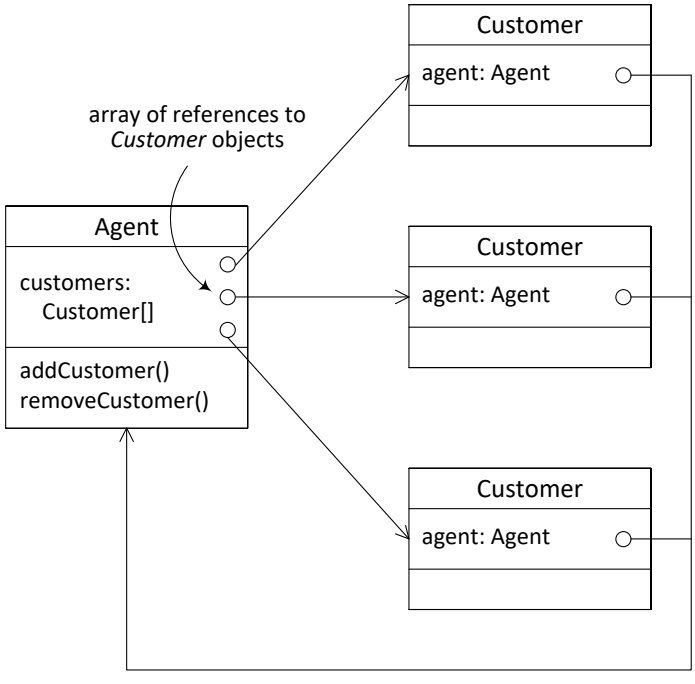

Figure 7.8: Implementation of the One-to-Many `supports` Association

7.5.3 Implementing Many-to-Many Associations

Consider the many-to-many association between `Supplier` and `Item` based on the `supplies` relationship shown in Figure 7.3. One way to implement the association is to maintain a collection of object references to `Item` objects in a related `Supplier` object, and to maintain a collection of object references to `Supplier` objects in a related `Item` object. Implementation of the `supplies` relationship using this approach is illustrated in Figure 7.9. The diagram shows a `Supplier` object that is related to five `Item` objects (only one `Item` object is drawn) and an `Item` object that is related to three `Supplier` objects (only one `Supplier` object is drawn).

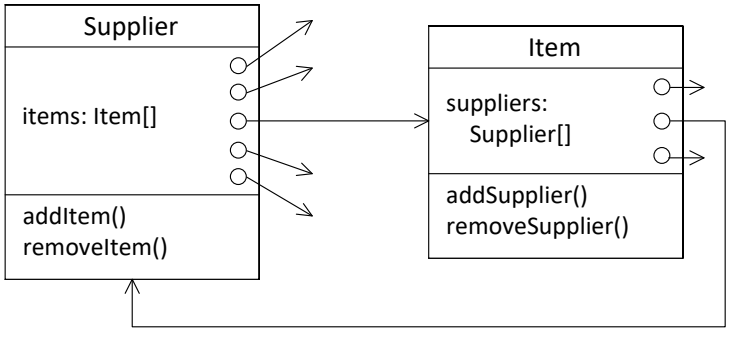

Figure 7.9: Implementation of the Many-to-Many `supplies` Association

Note that both `Supplier` and `Item` need to have methods that will enable objects to be added and removed from their respective collections as well as methods that allow the collections to be queried. Whenever a `Supplier` supplies a particular `Item`, the instance of the `Item` must be added to the collection of `Item`s in the given `Supplier` and the instance of the `Supplier` must be added to the collection of `Supplier`s in the given `Item`.

An alternative way to implement a many-to-many association is to use an *association class* (Rumbaugh et al. 1991; Booch, Rumbaugh, and Jacobson 1998). The use of an association class is discussed in the next sub-section.

7.5.4 Using an Association Class

It is possible to model and implement any relationship using an *association class*. An association class contains attributes and methods that pertain to the association between two or more objects. Thus, it can be viewed as an association that also has class properties. It is required in some many-to-many associations where *link attributes* (which pertain to an instance of a relationship) cannot be attached to one of the objects participating in the relationship without losing information (Rumbaugh et al. 1991).

The previous sections have shown that the implementation of a one-to-one association is different from the implementation of a one-to-many association, and this in turn is different from the implementation of a many-to-many association. If the multiplicity of the association between two classes changes over the lifetime of an application, at least one of the classes has to be modified. For example, if a one-to-one association changes to a one-to-many association, the object at the "one" side of the association must be re-designed with a collection of object references to the objects at the "many" end. Additional code must be written to handle retrieval, insertion, and deletion of

object references to and from the collection. Thus, a change to an association is expensive and requires the modification of existing classes.

Given the inconvenience and error-prone nature of making changes to existing classes, some authors suggest that *all* associations should be treated like many-to-many associations with link attributes (Date, 1995). Thus, association classes can be used to implement one-to-one, one-to-many, and many-to-many relationships. Figure 7.10 shows how an association class can be used to model the many-to-many association between a `Supplier` class and an `Item` class based on the `supplies` relationship.

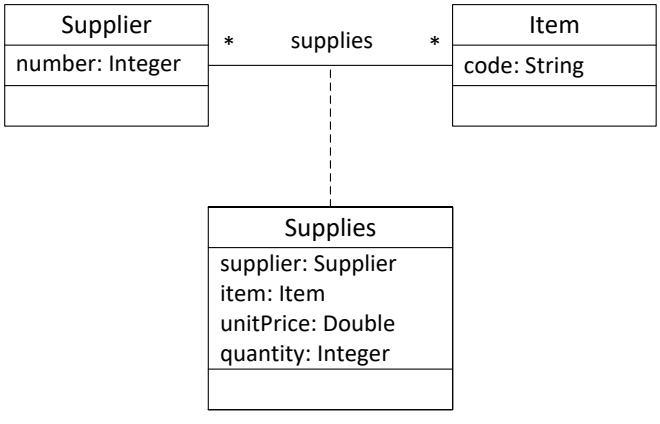

Figure 7.10: The `Supplies` Association Class

Figure 7.11 shows three cases of the `supplies` association where a `Supplier` object is linked to an `Item` object. Figure 7.11(a) shows that Supplier 10 supplies Item A100. Figure 7.11(b) shows that Supplier 10 also supplies Item B300. This is possible since a given `Supplier` can supply more than one `Item`. Figure 7.11(c) shows that Item A100 is also supplied by Supplier 20. This is possible since a given `Item` can be supplied by more than one `Supplier`.

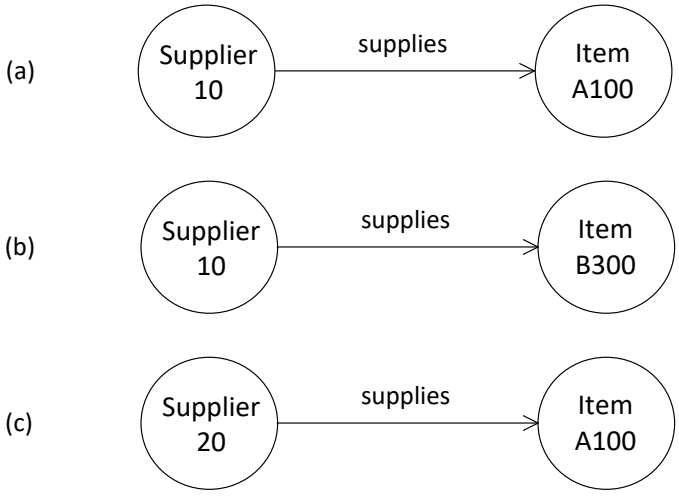

Figure 7.11: Cases of the `supplies` Association

Each case in Figure 7.11 is really an instance of the `supplies` association. Since an association class is now being used to represent the association, each case corresponds to an instance of the `Supplies` association class.

An instance of the `Supplies` association class may have attributes of its own. These are the link attributes mentioned earlier which store information on the "coming together" or linking of a `Supplier` instance and an `Item` instance. For example, the *quantity* of items supplied and the *unit price* at which they were supplied are link attributes. Note that the instance of the association class will always contain a reference to both the `Supplier` instance and the `Item` instance that it is linking together.

The instances of the association class must be stored somewhere. A good place to store the instances of the association class is the same place where the `Supplier` instances and `Item` instances are stored. Also, an appropriate collection should be used to store the instances of `Supplier`, `Item`, and `Supplies` based on the type of access desired. The class that manages the collections must also provide methods for client objects to query the collections. For example, a client may wish to find out how many times a particular `Item` was supplied by a particular `Supplier`. This query can be satisfied by traversing the collection of `Supplies` instances, checking to see which ones correspond to the given `Item` and the given `Supplier`.

The code below shows how the query can be implemented. It assumes that an array is used to store the collection of `Supplies` instances and that `numSupplies` is the amount of elements in the array.

```
public int numSupplied(Item item, Supplier supplier) {

   // supplies is an array of Supplies instances
   // numSupplies is the amount of elements in supplies

   int count = 0;

   for (int i=0; i<numSupplies; i++) {
      Supplies assoc = supplies[i];
                     // get instance of association

      Item assocItem = assoc.getItem();
                     // get Item object in association

      Supplier assocSupplier = assoc.getSupplier();
                     // get Supplier object in association

      if ((assocItem.getCode().equals(item.getCode())) &&
         (assocSupplier.getNumber() == supplier.getNumber()))
                     // check to see if supplier and item match

         count = count + 1;
                     // update count by 1
   }

   return count;
}
```

Note that when an association class is used, there is no longer a need to store a collection of references in `Supplier` or `Item` to maintain a one-to-one, one-to-many, or many-to-many association.

7.5.5 Implementing Aggregation and Composition

If an instance of a class `A` contains instances of a class `B`, there are various ways to implement the association, depending on the multiplicity of the association. If the instance of `A` contains only one instance of `B`, the association can be implemented by storing the reference to an instance of `B` in `A`. An instance variable of type `B` must first be declared in `A` as follows:

```
B b;
```

One of the methods of `A` is then responsible for creating `b`, or perhaps accepting a reference to `b` from an external source (see Table 7.2).

If an instance of `A` contains more than one instances of a class `B`, a collection is needed to hold all the instances contained. Similar to when implementing a one-to-many association, an array is a simple collection that can be used for this purpose. The array is declared as an instance variable of `A`:

```
B[] b;                  // elements to be stored are instances of B
```

The array is created as follows by some method of `A` (normally the constructor), or is supplied by an external source:

```
b = new B[size];        // create array of B with 'size' elements
```

Note that after creating the array, the instances of `B` do not yet exist and must be created individually. Thus, class `A` will have methods to add instances of `B` to the array as well as methods to delete instances and make queries on the objects contained.

Instead of using arrays, there are "collection" classes that are specially designed to store multiple objects such as the "parts" of an aggregation association or the objects at the "many" end of a one-to-many or many-to-many association. Collection classes facilitate different types of access (e.g., linear, tree-based) with different performance characteristics (e.g., $O(n)$, $O(n \log n)$). They belong to the Java *Collections* Framework and are discussed in Chapter 13.

Exercises

1. By giving suitable examples, explain what is meant by an association between two classes.

2. Explain what is meant by the multiplicity of an association. What are three typical multiplicities of a binary association? Give an example of each one.

3. By giving suitable examples, distinguish between aggregation and composition. Describe one way to implement each association.

4. Describe two ways in which a many-to-many association can be implemented.

5. Describe three ways in which an instance of a class `A` can get access to an instance of another class `B`.

6. In the context of an association class, explain what is a *link* attribute. Why does an instance of the association class always have a reference to the objects at both ends of the association?

7. Using appropriate classes, draw a UML diagram illustrating each of the following types of relationships: dependency, aggregation, and composition.