Chapter 4
Composition and Association

Pedagogic Motivations

- We introduce two new relationships that can exist between classes: composition and association. These relationships will allow us to build more elaborate domain models.
- We discuss the scope and lifetime of local and instance variables.
- Methods are introduced as a way of defining capabilities for objects.

Modeling with relationships

This chapter introduces two new relationships that can exist between objects. Unlike the dependency relationships we discussed in the previous chapter, which are fleeting in nature, the relationships discussed in this chapter persist for a longer period of time.

Another important concept introduced in this chapter is that of non-constructor methods. Each capability we identify for our objects is defined by a method in the class definition. Methods are also used to help set up relationships, such as association.

The lifetime of these relationships is captured using different types of variables. The short-lived dependency relationships we saw last chapter are created either without a variable (an instantiation dependency) or with a local variable (a local variable dependency). The longer-lived relationships of this chapter make use of a variable called an instance variable. This chapter includes a discussion of some important differences between local variables and instance variables.

Capabilities and Methods

An object communicates with another object by sending a message to it. The effect of the message is to trigger a capability on the receiving object (i.e. the receiving object performs the indicated capability). In Java, an object's response for a given capability is defined by a *method*. Another way of saying this is that a method defines the response of an object to a given message. When one object sends a message to another object, we typically say that the first object *invokes* (or *calls*) a method on the second object.

Example 1

Suppose we are modeling how a painter paints rooms in houses, with the aim of building an automatic house-painting robot. Included in this model may well be things like rooms, paint cans, paint brushes, colors and a model of the painter (which will be realized as a robot). The painter has paint cans containing paint of different colors, and different paint brushes, which it uses to apply color to the walls of a room.

We can send a message to the painter to change the color of the paint that is being used, perhaps by saying "Please change your paint color to *sweet blue*". Notice that not only do we need to tell the painter what we want done (change the paint color) but we also need to say to what color we want to change (*sweet blue*).

We can send another message to the painter, asking which color is currently being used to paint with, perhaps by saying "Pray tell, what paint color are you using now?" When we ask a question like this we expect to receive a response, maybe something like "*sea foam green*".

Example 2

One of the things we would like to do with a bank account is to check our balance. Suppose the bank account's response to the "what is my balance" query is defined by a method named `checkBalance`. The way we send a "what is my balance" query is by invoking (or calling) the `checkBalance` method on the bank account object.

Each capability we identify for an object during our modeling of a problem domain corresponds to a method; each method forms a part of the class definition for the object.

Method definition

Just like a class or a constructor definition, a *method definition* consists of two parts: a header and body. A method header has various parts, some required and some optional (to begin with we will ignore some of the optional parts; we will introduce them as needed). A method body consists of a sequence of statements which describe *how* to respond to the *invocation* of the method. The method body therefore provides procedural information[1] about what actions should be performed when the method is called. This will be realized by a sequence of statements about what activities the method should perform. Information about *what* the method should do is written as a *comment*.

Comments are written for human readers of a program. The compiler ignores comments, and they have no effect on the meaning of a program. In Java there are two forms of comments one can write. The first is a single-line comment, which can start at any point in a line (the start of the comment is indicated by double forward slashes, as in "`//`"), and continues to the end of a line. The second is a block comment. A block comment starts with "`/*`" and ends with "`*/`". A block comment includes all the text between "`/*`" and "`*/`".

The method header

The header of a method specifies certain information which is needed in order to invoke the method, such as the name of the method, whether the method requires information from the caller to do its job, and whether or not the method returns any value to its caller.

[1] Procedural information is "how to" information. Procedural information gives the steps an object must follow in order to carry out a method invocation.

Java requires that these bits of information be given in a particular order. Recall from the previous chapter our convention of writing in *italics* those pieces of Java syntax which are placeholders for information you provide. We will describe the structure of a method header as follows:

```
public type identifier (formal parameter-list)
```

There are three placeholder parts to this header definition: something called *type*, something called *identifier*, and something called *formal parameter-list*.

Let us deal with the most familiar one first: the *identifier* in the method header is the name that we will give to the method. The rules for naming methods are the same as those for naming classes. The conventions are a little different. Recall that rules must be followed, whereas conventions are guidelines which are a good idea to follow, but which you are not bound to follow.[2] The conventions of method naming are:

- Method names begin with a lower case letter.
- The underscore character '_', while a legal part of an identifier, is typically not used in a method name.
- Multi-word method names should capitalize every word after the first.[3]

The *type* part of the method header indicates what sort of value is returned from the method. We call this the *return type specification*.

Example 3

A capability of a bank account is to produce a monthly statement giving all the transactions that have occurred on the account since the previous statement. If we model this as a method named "getStatement", and we model a statement with a type named `Statement`, then the header of such a `getStatement` method might well look as follows:

```
public Statement getStatement()
```

So far the only mechanism we have seen to define a type is the class definition. The type defined by a class definition is sometimes called a *class type*, to distinguish it from other sorts of types. Sometimes a method does not need to return a value. In this case we use the keyword `void` as the return type specification. The keyword `void` indicates that the method does not return any value.

[2] If you saw the movie "Pirates of the Caribbean" you might recall that the "Pirate's Code" is described as "more of a guideline" than a rule: pirates should adhere to it, but pirates could do whatever they pleased – after all, they were pirates!

[3] This style of writing multi-word names, in which each new name begins with a capital letter, is referred to as "camel case", because the start of each word resembles the hump of a camel.

Example 4

Another method on a bank account, one which presumably does not return a value, is one which instructs the account to calculate the interest and fees for the month. A method such as this causes various transactions to occur on the bank account, but does itself not return a value. Suppose this method is called "calculateInterestAndFees". The method's header could look like this:

```
public void calculateInterestAndFees()
```

The *formal parameter-list* describes the values which need to be given to method for it to do its job. It is a comma-separated list of type and identifier pairs. Sometimes a method does not need any input values, in which case its formal parameter list is empty. Note that the formal parameter list is preceded by an opening parenthesis, '`(`', and is followed by a closing parenthesis, '`)`'. If the formal parameter list is empty, the parentheses are still present in the method header, but they do not have any contents. An empty formal parameter list therefore looks as follows: `()`.

Identifiers that are names of parameters follow the same naming conventions as local variables. This makes sense because, as we will learn, parameters are a particular form of local variable.

Example 5

Think back to the last time you visited a sit-down restaurant. Oftentimes your server will ask whether you want anything to drink before taking your order. If you do not want anything fancy to drink, the server will get some water. We can model this capability by defining a method named `getWater` in the `Server` class. This method returns a value (water in a glass); let us say that the value returned is of type `Water`. The method does not require any special information to carry out its task, so the method has an empty parameter list. The header of such a method looks like this:

```
public Water getWater()
```

Example 6

In a restaurant, the people who clean the tables after the patron has finished eating are called busboys and busgirls, or more generically bussers. Modeling a busser we could define a `Busser` class that has the capability to clean a table. In order to carry out this task appropriately and efficiently, a `Busser` must be told which table should be cleaned. A `cleanTable` method therefore requires a parameter, the table to be cleaned. A method such as this does not need to return and value to the caller when the activity is complete, and the method should have a `void` return type. The header of such a method might therefore look like this:

```
public void cleanTable (Table tableToBeCleaned)
```


Example 7

Recalling again about our Restaurant, we know that the Chef is able to make appetizers, main courses, and desserts from the menu. When a customer orders a specific main course from the menu, such as the Chicken Calabrese, the Chef is sent a message to make that dish. If another customer orders another specific dish, such as Surf & Turf, the Chef must receive a message which differs somehow from the one recevied for the Chicken Calabrese.

We could model the Chef with two methods, makeChickenCalabrese and makeSurfAndTurf, but this makes the Chef very rigid. The Chef becomes very tightly coupled with the foods that can be made. A better choice for modeling this interaction is to set up a parameterized "make main course" message, which can take different dish descriptions: "make main course". The two messages sent to the Chef are then: "make the Chicken Calabrese main course" and "make the Surf &Turf main course".

If we suppose that the description of the dish is of type MainCourseDescription and the value returned from the method is of type MainCourse, the header of this method could look like this:

```
public MainCourse makeMainCourse(MainCourseDescription selection)
```

Example 8

One more example comes from our Restaurant domain. When the last employee leaves the restaurant at night they must lock up the restaurant. This is a capability which does not require any additional information for someone to do it (i.e. no parameters are needed), nor does it require any information to be returned (i.e. it has a `void` return type). The header of such a method might look like this:

```
public void lockUp ()
```

The method body

The second part of a method definition, the method body, consists of a sequence of statements which describe how to respond to the invocation of the method. The method body begins with an opening brace, '`{`', and ends with a closing brace, '`}`'. We will see examples of method bodies later in this chapter.

Constructors are methods

We first began looking at capabilities in chapter 2 when we created constructors. At the time, we said that the constructor was a special type of capability that helps initialize the state of an object. In this chapter, we learned that the formal specifications of capabilities are methods. Constructors are in fact methods, but they are special in many ways. One way in which they are special is that their name must be the same as the name of their class. This is not the case for other methods. In fact, non-constructor methods should be named differently from their classes.

Another way in which constructors differ from other methods is that they are not permitted to have any return type specification. The return type of a constructor is not `void`, or any other type. Constructors simply do not have a return type in their headers at all.

Finally, constructors can be invoked only together with the `new` operator. Regular methods do not need to use the keyword `new` to be invoked. In fact, if you try to use `new` with a regular method, your code will not compile. In the same way, you can not use the dot operator to call the constructor.

Two methods cannot have the same header

In Java, as in many other programming languages, two methods cannot have the exact same method header. This means that no two methods can have the same return type, method name, and parameter lists.[4] The reason for this is similar to the reason why no two classes can be named the same. If there were two methods that had the exact same header and we tried to call one of the methods, how would the computer know which method we wanted to be executed? How would the computer resolve the ambiguity? Since there is no easy way to resolve the ambiguity, it is prohibited for two methods to have the same method header.

Variables revisited

In the last chapter we learned about local variables. We now learn that there are many different sorts of variables (local variables, parameters, and instance variables). We discuss properties of variables, scope and lifetime, which distinguishes between them.

Scope and Lifetime of Variables

Variables have many properties. We have come across three of these properties when discussing local variables in the last chapter. There we learned that a variable has a type, a name, and a value. The type and name of a variable are required components of a variable declaration. The value of a variable is set using an assignment statement.

Every variable also has a *scope* and a *lifetime*. The *scope* of a variable is the region of the program text in which the variable is accessible. The *lifetime* of a variable is the period of time during the execution of a program during which it exists.

To make these concepts more concrete, let us consider the scope and lifetime of local variables. After we introduce *instance variables*, a new sort of variable we are introducing in this chapter, we will then discuss their scope and lifetime properties.

A local variable, as defined in the previous chapter, is a variable declared in the constructor of a class. In fact, local variables are variables declared in any method of a class.

The scope of a local variable extends from its point of declaration to the end of the body in which the declaration occurs. Thus, in the following example, the scope of the variable `ourFirstTerrarium` is the shaded region:

[4] Java does not allow methods to be differentiated solely on the basis of their return type. Java pays attention only to the name and the parameter list.


```
package chapter3;

public class EcoSystem {
      public EcoSystem() {
            chapter1.Terrarium ourFirstTerrarium;
            ourFirstTerrarium = new chapter1.Terrarium();
            ourFirstTerrarium.add(new chapter1.Ant().start());
            ourFirstTerrarium.add(new chapter1.Ant().start());
      }
}
```

In particular, the variable cannot be used before its declaration – the following code cannot be compiled, and the compiler flags an error on the shaded line:

```
package chapter3;

public class EcoSystem {
      public EcoSystem() {
            ourFirstTerrarium = new chapter1.Terrarium();
            chapter1.Terrarium ourFirstTerrarium;
            ourFirstTerrarium.add(new chapter1.Ant().start());
            ourFirstTerrarium.add(new chapter1.Ant().start());
      }
}
```

Turning to lifetime, a local variable comes into existence when the method it is declared in is called, and ceases to exist when execution of the method is finished.[5] It is important to realize that scope and lifetime are distinct. The scope of a variable determines where in the text of a program it can be used. The lifetime of a variable determines the points in time during execution of a program that the variable actually exists in the memory of the machine. A variable may have several lifetimes during the execution of a program: each time the method is called, the variable is created and then destroyed.

The scope of a parameter is the entire method body. The lifetime of a variable extends from the method call until the end of the execution of the method.

Instance variables

To realize the composition relationship we need to declare what is called an *instance variable*. Instance variables are so named because they are variables that are associated with instances of a class; each instance of a class has its own set of instance variables. The form of an instance variable declaration is:

```
access-control-modifier type identifier ;
```

[5] Although there can be exceptions, generally speaking space for all local variables is set aside when a method is called, even if the declaration occurs later on in the method.

Recall from chapter 2 what an *access control modifier* is. The keyword `public` is an example. When applied to a method, this keyword permits access to the method from outside the class body. It is inappropriate to use the `public` modifier with properties, however, because we do not want properties to be generally accessible. Instead we want to restrict access to properties as much as possible. Java has an access control modifier, `private`, which restricts access to the element so modified to the class in which it is defined. [6] Thus, our instance variable declarations will have the following form:

```
private type identifier ;
```

Why do we want instance variables to be `private` rather than `public`? And why do we declare methods to be `public` rather than `private`? Methods are declared `public` because it is via methods that objects communicate with each other. If all methods of a class were marked as `private` then no communication between objects could occur. The properties of an object must remain under the exclusive control of the object, however.

Example 9

When you open an account with a bank, the bank keeps track of how much money is in your account, and defines certain operations you can perform to change the balance in the account. You can, for example, deposit money into the account, which increases the amount of money in the account. You can also withdraw money from the account, which decreases the amount of money in the account. The bank ensures that the only way that you can modify the balance in your account is to use the deposit and withdrawal methods. You cannot simply set your account balance to an arbitrary value (and neither can anybody else). The balance property of the bank account is kept private, and the ability to modify the account balance is provided via the deposit and withdrawal methods. These methods can impose conditions on their use. The withdrawal method, for example, can ensure that withdrawals are only permitted when there is enough money in the account.

How do the scope and lifetime of instance variables compare to that of local variables? Recall that the scope of a local variable does not extend beyond the method in which it is declared. The scope of an instance variable is the entire body of the class definition in which the instance variable is declared, regardless of where in the class definition the instance variable declaration occurs. The practical effect of this is that we can refer to an instance

[6] This idea that the internal data of an object should be kept private is called *information hiding*. It goes hand-in-hand with the idea of *encapsulation*, which refers to the gathering together of related pieces of information in one place. These are general software development ideas. In object-oriented terms objects are encapsulations of their state; since the state of an object should be private we apply the principle of information hiding and mark each instance variable as `private`.

variable in any method in the class definition, unlike local variables which can only be used within the method in which they are declared.

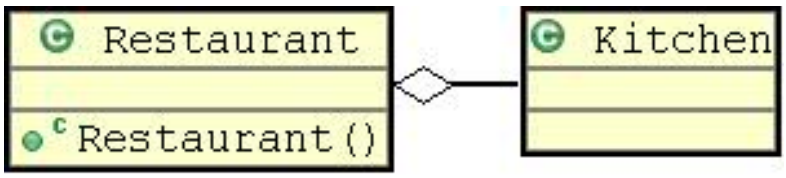
The lifetime of a local variable is the duration of a method call. An instance variable comes into existence when the object of which it is a part is created, and exists as long as its object exists.[7] This means that as long as an object exists, its instance variables also exist. This is important because the values of instance variables persist between method calls, in contrast to the values of local variables, which do not persist.

Motivating the Composition Relationship

Let us think about our restaurant example in a bit more detail. In this domain there are many different relationships which exist between elements in the domain. Some relationships are relatively fleeting, while other relationships are longer term. The relationship that a patron has with a hostess is quite short-lived: it exists from the time that the patron shows up at the restaurant and ends when they have been seated at a table. The relationship between the restaurant and its kitchen is quite different. A restaurant without a kitchen just is not a restaurant! In modeling terms, we want to capture that a kitchen is an integral part of a restaurant. We take this to mean that a kitchen is composed of a restaurant, and that a restaurant's kitchen must exist when the restaurant first opens its doors for business. We think of this relationship a *whole-part* relationship because it models the relationship between a whole (the restaurant) and one of its integral parts (the kitchen). It is called the *composition* relationship. Less formally it is referred to as the *"has a"* relationship.

The Composition Relationship in UML

In UML class diagrams the composition relationship is shown using a solid line connecting the two classes involved in the relationship. The line is marked with a diamond shape at the "whole" class. This is shown in the diagram below.



Recall that our relationships are directional. In the composition relationship we call the whole class the source and the part class the target. In this example the `Restaurant` class is the source class of the relationship, and the `Kitchen` class is the target class of the relationship.

The Composition Relationship in Code

We want the Restaurant to have the Kitchen as a property. We start by declaring an instance variable of the correct type in the Restaurant class:

[7] Unlike some other languages, notably C++, Java does not put control of object destruction into the hands of the programmer. Instead Java, like many modern languages, has a "garbage collection" facility which reclaims the memory occupied by objects once they are no longer needed.


```
package chapter 4;

public class Restaurant {

    private Kitchen _kitchen;

    public Restaurant() {
    }

}
```

Here we see another naming convention: the names of instance variables follow the same general naming conventions as names of methods, with the exception that their first character is an underscore character '_'.

Since we want to ensure that whenever a new `Restaurant` object is created it is guaranteed to have a `Kitchen`, we can assign a new `Kitchen` object to this variable in a constructor of the `Restaurant` class:

```
public Restaurant () {
      _kitchen = new Kitchen();
}
```

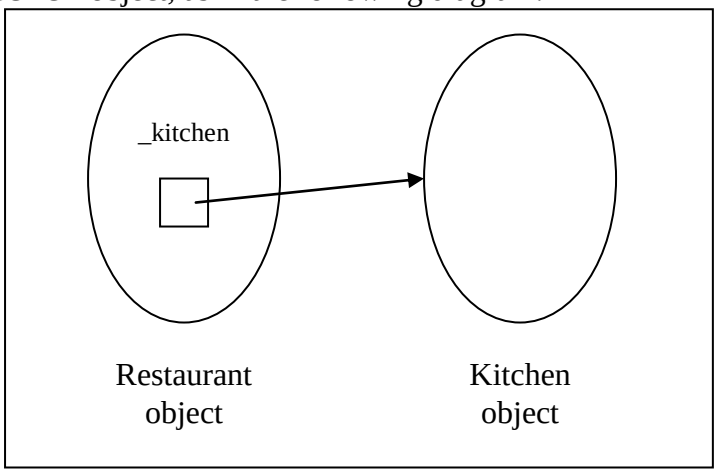
Returning one last time to our example and putting all the pieces together, here is what the complete definition for the class `Restaurant` looks like:

```
package chapter4.restaurant;

/**
 * @author alphonce
 *
 * This class models that a every Restaurant must
 * also have a Kitchen; this is modelled using a
 * composition relationship.
 */
public class Restaurant {

      /**
       * The _kitchen of a Restaurant object.  The _kitchen is
       * a **part** of a Restaurant.
       */
      private Kitchen _kitchen;

      /**
       * The constructor of the Restaurant objet ensures it has a
       * Kitchen object.
       */
      public Restaurant() {
            _kitchen = new Kitchen();
      }
}
```

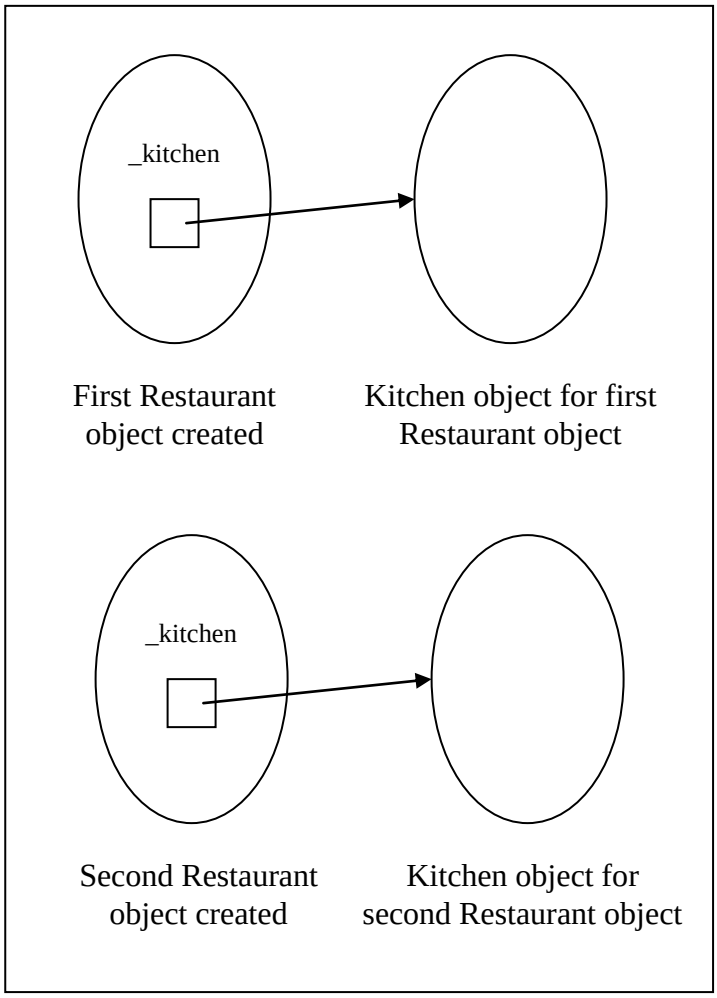

An Object Model

We now introduce an informal notation to help us visualize and understand the *object model* of the programs that we write. Recall that we use UML class diagrams to exhibit the class structure of programs. Class diagrams are static, in the sense that they reflect only the structure of a program *prior* to runtime. Models, as we have introduced them, are runtime models: they are systems of interacting objects. An object model is a visual depiction of what happens at runtime, during execution of a program.

In these object models we will draw objects as ovals, and variables as squares. For example, suppose we instantiate the `Restaurant` class, as a result of executing the code `new Restaurant()`. Two objects are created as a result: the first an instance of the `Restaurant` class, while the second is an instance of the `Kitchen` class. The latter object is created in the `Restaurant` class' constructor. We can show these two objects, the `_kitchen` instance variable of the `Restaurant` object, and its value as a reference to the `Kitchen` object, as in the following diagram:



Now suppose we had instantiated the `Restaurant` class twice. In this case two `Restaurant` objects would be created, *each with its own `Kitchen` object!* This can be shown as in the following diagram:



Here we also see the effect of the composition relationship: every time a new `Restaurant` object is created, a new `Kitchen` object is created as well. The lifetime of the `Kitchen` object is tied to the lifetime of the `Restaurant` object. We sometimes say that the object playing the role of the whole has responsibility for creating the object playing the role of the part.

Exercise

Draw an object diagram showing the objects created when instantating the following class:

```
public class Meal {
    private Appetizer _appetizer;
    private MainCourse _mainCourse;
    private Dessert _dessert;
    public Meal() {
        _appetizer = new Appetizer() ;
        _mainCourse = new MainCourse();
        _dessert = new Dessert();
    }
}
```

Motivating the Association Relationship

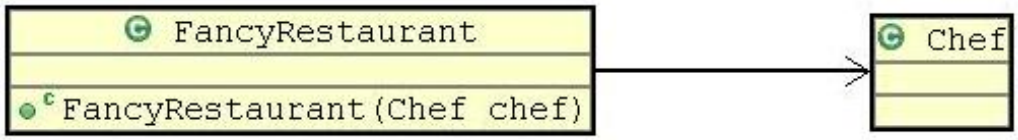
The composition relationship involves an instance variable and therefore enables communication between the whole object and the part object. Composition models a very particular type of relationship, one in which the lifetimes of the two objects involved are intimately tied together. While composition is a common relationship between objects, it is not always the type of relationship that exists between objects. Another common relationship is called *association*. The association relationship is used to model that one object can communicate with another object, but with no commitment that the lifetimes of the objects are tied to each other.

The Association Relationship in UML

We argued above that it could make sense to model the relationship between a restaurant and its kitchen as a whole-part (or "composition") relationship. The relationship between a restaurant and its chef is different. A restaurant typically must have a chef when it opens its doors, but the restaurant does not build (or create) its chef in the way it builds (or creates) its kitchen. Moreover, a restaurant may change its chef over time. The relationship between a restaurant and its chef is an example of an association relationship.

There are a few different ways in which an association relationship can be implemented. We will discuss the possibilities in turn. In all cases an association is drawn in a UML class diagram in the same way: a line solid line is drawn between the source class and the target class, with an open arrowhead pointing to the target class.

The association relationship between a `FancyRestaurant` and its `Chef` is drawn as follows:



The Association Relationship in Code (variation 1)

The implementation of the association relationship is similar in one respect to that of the composition relationship: it involves an instance variable. Recall that in the case of composition, the constructor of the source class took responsibility for creating an instance of the target class. In the case of the `Resaurant` class constructor we had:

```
public Restaurant () {
      _kitchen = new Kitchen();
}
```

In the case of an association, we do not want the source class to have the responsibility of creating an instance of the target class. The purpose of the association relationship is to enable communication from instances of the source class to instances of the target class without having the responsibility of creation. This can only occur if the instance of the source class has a reference to an instance of the target class. This means we have to make sure there is an assignment to a suitable instance variable, declared to be of the target class' type.

One straightforward way to do this is to simply set up the constructor to accept a reference to an instance of the target class as a parameter, and to assign this reference to the instance variable. Here is the basic setup needed for the `FancyRestaurant` constructor to support this::

```
public FancyRestaurant (Chef chef) {
      _chef = chef;
}
```

Putting all the pieces together into a class definition gives us this:

```
package chapter 4 ;

public class FancyRestaurant {
    private Chef _chef;

    public FancyRestaurant (Chef chef) {
      _chef = chef;
    }
}
```


Mutator Methods

If we want to include the possibility that the chef of a restaurant can change over time, we can include a method which we can call to assign a new value to the `_chef` instance variable. This is our first example of a non-constructor method:

```
public void setChef(Chef chef) {
      _chef = chef;
}
```

A method such as this, whose only job it is to assign a new value to an instance variable, is called a *mutator* method (because it mutates, or changes, the value of an instance variable). A mutator method is sometimes also called a "setter" method (because it sets the value of an instance variable).

Notice how the parameter `chef` is used to temporarily hold the argument value until it is assigned to the instance variable `_chef`.

Accessor Methods

If we want to include the capability of a restaurant to tell us who its current chef is, we can include a method which we can call to return the current value to the `_chef` instance variable.

```
public Chef getChef() {
      return _chef;
}
```

A method such as this, whose only job it is to provide the current value to an instance variable to the caller of the method, is called an *accessor* method (because it accesses, or provides, the value of an instance variable). An accessor method is sometimes also called a "getter" method (because it gets the value of an instance variable).

Revisiting the Association Relationship

We now have the ability to establish an association relationship in two different ways. One way is to set it up the relationship in the constructor of the source class. This requires that the target object exist before the source object, so a reference to it can be passed in through the parameter of the source object's constructor.

Another way is to use a mutator method to allow the two objects to be associated at any time after the source object is created. In this way the target object can be created after the source object comes into existence.

The Association Relationship in Code (variation 2)

This is the basic form of the association relationship, established using a contstructor for the class:


```
package chapter 4 ;

public class FancyRestaurant {
    private Chef _chef;

    public FancyRestaurant (Chef chef) {
      _chef = chef;
    }
}
```

This is the basic form of the association relationship, established using a mutator method:

```
package chapter 4 ;

public class FancyRestaurant {
    private Chef _chef;

    public FancyRestaurant () {
    }

    public setChef(Chef chef) {
      _chef = chef;
    }
}
```

Of course, it is possible to combine these two approaches so as to require that the initial value of the variable be specified via the constructor, yet still permit its value to be changed via the mutator:

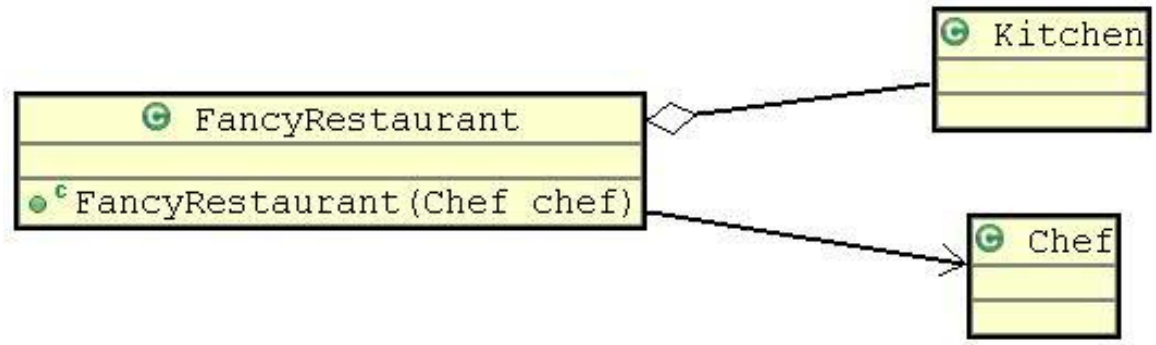
```
package chapter 4 ;

public class FancyRestaurant {
    private Chef _chef;

    public FancyRestaurant(Chef chef) {
      _chef = chef;
    }

    public setChef(Chef chef) {
      _chef = chef;
    }
}
```

Multiple relationships

A fancy restaurant presumably has both a kitchen and a chef. Can a class enter into more than one relationship? Certainly. Here we show the `FancyRestaurant` class defined so that it is composed with a `Kitchen` and is associated with a `Chef`:


```
package chapter4.restaurant;

/**
 * @author alphonce
 *
 * This class models that a every FancyRestaurant must
 * be composed of a Kitchen, and must know a Chef; the relationship
 * with the Chef is modeled using an association relationship.
 */
public class FancyRestaurant {

	/**
	 * The _kitchen of a FancyRestaurant object.  The _kitchen is
	 * a **part** of a FancyRestaurant.
	 */
	private Kitchen _kitchen;

	/**
	 * The _chef of a FancyRestaurant object.  The _chef is known
	 * to a FancyRestaurant.
	 */
	private Chef _chef;

	/**
	 * @param chef (of the FancyRestaurant)
	 */
	public FancyRestaurant(Chef chef) {
		_kitchen = new Kitchen();
		_chef = chef;
	}
}
```

The UML class diagram below shows the relationships that the `FancyRestaurant` class has with the two classes `Kitchen` and `Chef`.



Summary

Two new relationships were presented: composition and association. Composition models a whole-part relationship between objects. This relationship requires that the part object is created by the whole object when it (the whole object) is created. The association relationship implies no such lifetime dependency between objects, but simply requires that the source object maintain a reference to the target object. This reference permits the source

object to send messages to the target object. Recall that references are the way that we can hold onto objects. We create variables and in the case of these relationships, instance variables to hold onto our objects.

Two important properties of variables, their scope and lifetime, were introduced. The scope of a variable is the region in the source code file in which a variable is accessible. Scope is therefore a static property of a variable. The lifetime of a variable is the period of time during the execution of a program when it exists, from when it is created as an instantiation of a class to when it is destroyed (which we can take to be the point in time when the program terminates).

Case Study

Stay tuned…

Chapter Wrap-Up

Learning Objectives

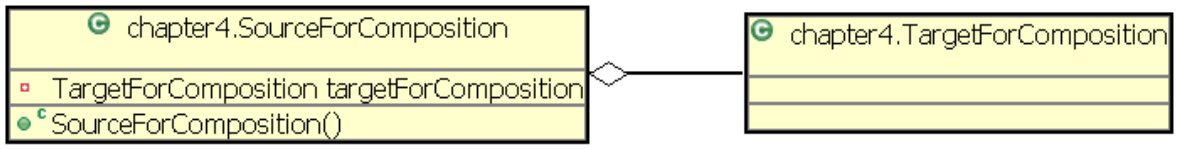
At the end of this chapter students should be able to:

- Define accessor and mutator methods.
- Declare instance variables.
- Assign values to instance variables.
- Differentiate between an object and a reference to an object.
- Differentiate between an expression and its value.
- Understand what a composition relationship models.
- Understand what an association relationship models.
- Implement the composition relationship in Java.
- Implement the association relationship in Java.

Relationships Covered

Composition

Informal Name: *has-a*
First introduced in: Chapter 4 – page 9
UML representation:



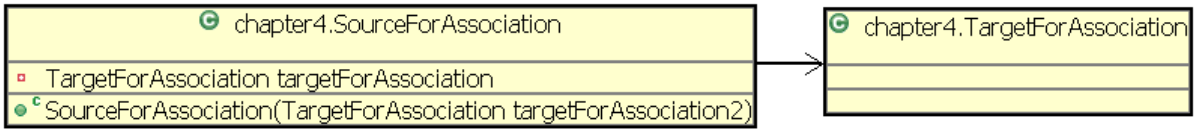
In code: (from above diagram)

```
package chapter4;

public class SourceForComposition {
	private TargetForComposition _targetForComposition;
	public SourceForComposition() {
		_targetForComposition = new TargetForComposition();
	}
}
```

Note: There is no code inside the class `TargetForComposition` to indicate the relationship

Association

Informal Name: *knows-a*
First introduced in: Chapter 4 – page 13
UML representation:



In code – variation 1: (from above diagram)

```
package chapter4;

public class SourceForAssociation {
	private TargetForAssociation _targetForAssociation;

	public SourceForAssociation(TargetForAssociation tfa) {
		_targetForAssociation = tfa;
	}
}
```


Note: There is no code inside the class `TargetForAssocation` to indicate the relationship

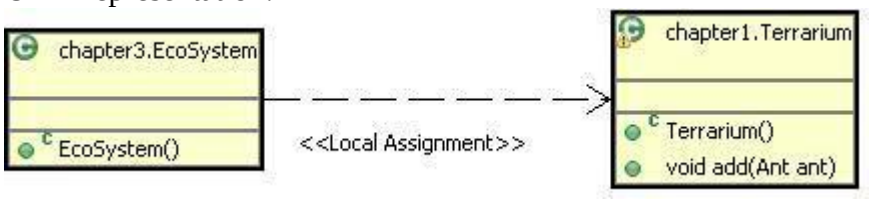
In code – variation 2:

```
package chapter4;

public class SourceForAssociation {
     private TargetForAssociation _targetForAssociation;

     public SourceForAssociation() {

     public void setTarget(TargetForAssociation tfa) {
          _targetForAssociation = tfa;
     }
}
```

Note: There is no code inside the class `TargetForAssocation` to indicate the relationship

Local Variable Dependency

Informal Name: *uses-a*
First introduced in: Chapter 3 – page **Error! Bookmark not defined.**
UML representation:



In code: (from above diagram)

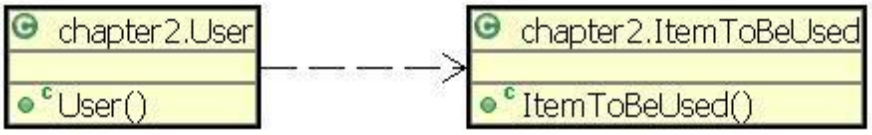
```
package chapter3;

public class EcoSystem {
     public EcoSystem() {
          chapter1.Terrarium _terrarium = new chapter1.Terrarium();
     }
}
```

Note: There is no code inside the class `chapter1.Terrarium` to indicate the relationship.

Instantiation Dependency
Informal Name: *uses-a*

First introduced in: Chapter 2 – page **Error! Bookmark not defined.**

In UML representation:



In code: (from above diagram)

```
package chapter2;

public class User {
     public User() {
          new ItemToBeUsed();
     }
}
```

Note: There is no code inside the class `ItemToBeUsed` to indicate the relationship.

Keywords Covered

New in this chapter:

`private` (page 8)

`void` (page 3)

Previously Covered:

`class` (**Error! Bookmark not defined.**)

`new` (**Error! Bookmark not defined.**)

`null` (**Error! Bookmark not defined.**)

`package` (**Error! Bookmark not defined.**)

`public` (**Error! Bookmark not defined.**)

Naming Conventions Covered

Classes – Begin with an upper case letter. The first letter of each subsequent word in class name is capitalized.

Instance variables – Begin with an underscore character. First letter of each subsequent word in instance variable name is capitalized.

Local variables – Begin with a lower case letter. First letter of each subsequent word in variable name is capitalized.

Methods – Begin with a lower case letter. The first letter of each subsequent word in method name is capitalized.

Packages – Are written entirely in lowercase. Package names in nested packages are separated by dots.

Vocabulary & Programming Terms Covered

New in this chapter:

accessor (15)

calling [a method] (1)

class type(3)

comment (2)

instance variable (7)

invoking [a method] (1)

lifetime (6)

method(1)

method definition (2)

mutator (15)

parameter list (4)

return type specification (3)

scope (6)

type(3)

void (3)

Previously Covered:

abstraction (**Error! Bookmark not defined.**)

access control modifier (**Error! Bookmark not defined.**)

assignment operator (**Error! Bookmark not defined.**)

bytecode (**Error! Bookmark not defined.**)

calling [a constructor] (**Error! Bookmark not defined.**)

capability (**Error! Bookmark not defined.**)

Church-Turing thesis (**Error! Bookmark not defined.**)

class (**Error! Bookmark not defined.**)

class body (**Error! Bookmark not defined.**)

class diagram (**Error! Bookmark not defined.**)

class header (**Error! Bookmark not defined.**)

compiler (**Error! Bookmark not defined.**)

conceptual model (**Error! Bookmark not defined.**)

constructor (**Error! Bookmark not defined.**)

constructor body (**Error! Bookmark not defined.**)

constructor definition (**Error! Bookmark not defined.**)

constructor header (**Error! Bookmark not defined.**)

domain (**Error! Bookmark not defined.**)

edit-compile-run cycle (**Error! Bookmark not defined.**)

editor (**Error! Bookmark not defined.**)

executable model (**Error! Bookmark not defined.**)

expression (**Error! Bookmark not defined.**, **Error! Bookmark not defined.**)

IDE (**Error! Bookmark not defined.**)

identifier (**Error! Bookmark not defined.**)

instantiate [a class] (**Error! Bookmark not defined.**)

interactions pane (**Error! Bookmark not defined.**)

iterative process (**Error! Bookmark not defined.**)

invoking [a constructor] (**Error! Bookmark not defined.**)

Java Virtual Machine (JVM) (**Error! Bookmark not defined.**)

javac (**Error! Bookmark not defined.**)

keyword (**Error! Bookmark not defined.**)

local variable (**Error! Bookmark not defined.**)

model (**Error! Bookmark not defined.**)

naming conventions (**Error! Bookmark not defined.**)

naming rules (**Error! Bookmark not defined.**)

object (**Error! Bookmark not defined.**)

package (**Error! Bookmark not defined.**)

package declaration (**Error! Bookmark not defined.**)

problem domain (**Error! Bookmark not defined.**)

process (**Error! Bookmark not defined.**)

property (**Error! Bookmark not defined.**)

property value (**Error! Bookmark not defined.**)

reference (**Error! Bookmark not defined.**)

relationship (**Error! Bookmark not defined.**, **Error! Bookmark not defined.**)

semantics (**Error! Bookmark not defined.**)

service (**Error! Bookmark not defined.**)

source code (**Error! Bookmark not defined.**)

state (of an object) (**Error! Bookmark not defined.**)

statement (**Error! Bookmark not defined.**)

syntax (**Error! Bookmark not defined.**)

top-level object (**Error! Bookmark not defined.**)

UML [Unified Modeling Language] (**Error! Bookmark not defined.**)

uncomputability (**Error! Bookmark not defined.**)

variable (**Error! Bookmark not defined.**)

Design Patterns Covered

We have not covered any design patterns so far in this text.

