Chapter 3
Naming Objects: Variables and References

Pedagogic Motivations

- Naming objects so that we can refer to them more than once is a very important concept. This chapter studies naming in detail, explaining the difference between an object and a reference to an object, and the difference between anonymous and named references.
- The local variable dependency is introduced.

Introduction

One of the first things two people exchange when they meet is names. People do this in order to have a way of referring to each other. In fact, naming is so useful that people give names to many things in their environment, not only people. Pets are given names, as are cars, houses, even pieces of furniture. Once an object has been named it can be referred to and talked about.

Objects and references

Building a model involves more than just identifying objects. We said early on in our study of object oriented modeling that object oriented models consist of collections of *interacting* objects. Remember that objects interact with each other by sending messages to each other. In order for one object to send a message to another object, the object sending the message must be able to refer to the object that the message is destined for. Our ability to build an object oriented system is severely limited by our inability to refer to an object after it's been created.

Let us say, quite informally for the time being, that if one object can refer to another object then the former has a *reference* to the latter. Having a reference to an object is a central aspect of object communication. A goal of this chapter is to explain how we can associate names with objects in programming languages. There are many different ways in which one object can come to have a reference to another object. The different ways in which the reference is established correspond to different relationships. In this chapter we introduce one new relationship that can occur between objects in a model, *local variable dependency*.

Referring to Objects

We have seen how to create an object and invoke one of its capabilities. An example we showed in an earlier chapter was the following:

```
new chapter1.Terrarium().add(new chapter1.Ant())
```

We know that evaluating this expression results in two new objects being created: a Terrarium object and an Ant object. The add capability of the Terrarium adds the object it is given in its argument list to the Terrarium environment, so that the Ant appears within the graphical representation of the Terrarium.

Suppose we wanted to have more than one Ant in the Terrarium at a given time. With what we know so far about programming languages, we cannot do this! We can certainly create more Ant objects – each time we evaluate the expression `new chapter1.Ant()` a new Ant object is created, so that's not the problem. The problem is that we cannot keep adding Ant objects to the Terrarium object we already have, because we have no way of referring to that Terrarium object more than once. In particular, we cannot repeat the same expression, as in,

```
new chapter1.Terrarium().add(new chapter1.Ant())
```

because then yet another Terrarium is created to hold the new Ant, rather than putting the Ant into the Terrarium we already have (try it)! In effect this expression says "create a new Ant object and put it into a new Terrarium object". What we want to the able to say is "create a new Ant object and put it into the Terrarium object we created before". To express "the Terrarium object we created before" we need a name.

Let us suppose we were able to name the Terrarium object we created in the first expression `ourFirstTerrarium`. In this case we could achieve what we want by writing

```
ourFirstTerrarium.add(new chapter1.Ant())
```

So what can we do? In order to be able to refer to one and the same object across several method invocations, we need to *name* that object. In Java we use a *variable* to associate a name with an object reference.

Declaring variables

Java requires that, before we use a variable, we say what *type* of object the variable will refer to. What is a type? For now, we can equate "type" with "class", since classes are the only sort of type we have seen so far. Later on we will learn that there are different sorts of types, and at that point we will need to make a distinction between the broader concept of type and the narrower concept of class.

We express the type of a variable in a *variable declaration*. We will come across variable declarations in a variety of situations in Java. At its most basic, a variable declaration consists of a type and an identifier (the *name* of the variable):[1]

[1] Languages differ in whether they require programmers to declare variables. Statically typed languages require that the types of all variables be known at compile time, whereas dynamically types languages only require that types be known at runtime. There are advantages and disadvantages to both static typing and dynamic typing, but a discussion of the pros and cons are best left for a programming languages course to tackle.

```
type identifier
```

Where in a class definition does a variable declaration belong? The answer to this question really depends on what role we want the variable to play in the class. To start with we will discuss variables which are declared inside a constructor for a class.

Declaring local variables

A declaration of the variable `ourFirstTerrarium` to be of type `chapter1.Terrarium` takes the following basic form:

```
chapter1.Terrarium ourFirstTerrarium
```

The exact form of the declaration depends on where it appears. The first place we will explore variable declarations is within a constructor for the class. In this case the variable is known as a *local variable*. The simplest form of a local variable declaration is,

```
type identifier;
```

The convention for a local variable name is that it should begin with a lower case letter, and the first letter of each subsequent word in method name is capitalized.

The ';' indicates the end of the variable declaration, in the same way that a period indicates the end of a sentence. Continuing our earlier example, the complete declaration of the variable `ourFirstTerrarium` to be of type `chapter1.Terrarium` as a local variable is:

```
chapter1.Terrarium ourFirstTerrarium;
```

Objects and References

Now we have a variable! A variable is used to create a name-value association. We gave the variable a name in its declaration, but what about a value? In other words, how to we make `ourFirstTerrarium` refer to a specific `chapter1.Terrarium` object? To answer this question, let us review a little.

We have learned that we can create a new object by instantiating a class. We do this by using the `new` operator together with a constructor for the class we wish to instantiate, as in:

```
new chapter1.Terrarium()
```

This is an example of something called an *expression*. An expression can be evaluated. You are probably used to seeing expressions like `3+4` and `7*2`. These are arithmetic expressions. They can be evaluated to produce the values `7` and `14` respectively.

What is the value of an expression like `new chapter1.Terrarium()`? The obvious answer is that it is an object, and this is *almost* correct. When evaluated, it does result in the creation of an object, but the value of this expression is something called a *reference* to an object.

What is the difference between an object and a reference to an object? Consider the difference between a house and the address of a house. We can draw an analogy between an *object* and *house* on the one hand, and a *reference to an object* and the *address of a house* on the other. The address of a house can be used to refer to a house without having to go to the house. For example, I can tell you that when I was young I lived at 9 West View Road. Having the address makes it easier to refer to the house from lots of different places – it enables me to refer to the house without actually being at the house. Similarly a reference to an object allows one to refer to an object without having to be "at the object" (in a sense to be made more precise later on).

To set up the name-value association of a variable, we *assign* a reference to the variable (the reference is to the object we wish to name). Assignment of a value to a variable is accomplished using an assignment statement, whose general form is:

```
idenfifier = expression;
```

"=" is called the *assignment* operator. The assignment operator evaluates the expression on its right (called the RHS, short for Right Hand Side) and assigns the resulting value to the variable on its left (called the LHS, short for Left Hand Side). The identifier must therefore be the name of a variable, and the value of the expression must be of the type as the declared type of the variable being assigned to.

To associate a `chapter1.Terrarium` object with the variable `ourFirstTerrarium`, we write:

```
chapter1.Terrarium ourFirstTerrarium;
ourFirstTerrarium = new chapter1.Terrarium();
```

What is an expression?

We had said that something like new chapter1.Terrarium() is an expression, whose value is a reference to an object. But is every expression of this form? No – there are many ways to form expressions in Java. In fact, a variable can be an expression! The value of a variable is the reference assigned to it.[2] This is why we can write things like:

```
ourFirstTerrarium.add(new chapter1.Ant())
```

Combining declarations and assignments

Like many programming languages, Java allows us to combine the declaration of a variable with the assignment of an initial value to it. An initializing declaration has the following form:

```
type identifier = expression;
```

A concrete example would be,

```
chapter1.Terrarium ourFirstTerrarium = new chapter1.Terrarium();
```

The `null` reference

Once we have the notion of a reference to a particular object we can talk about a reference which does not refer to any object. This is the address equivalent of an address which does not correspond to any real house. For example, there was no 0 West View Road on the road where I grew up. In terms of object references there is only one reference which does not refer to an object; it is called `null`. The Java compiler gives the default value of `null` to any variable not explicitly assigned a value.[3,4]

---

[2] Things are not quite as simple as this. In fact, when the variable appears to the left of the assignment operator, '=', the value of the variable is its location in the memory of the computer. This value is known as the variable's "l-value" (the value we are referring to when the variable occurs to the left of the assignment operator. In other situations the value of the variable is its "r-value" (the value we are referring to when the variable occurs to the right of the assignment operator).

[3] We will learn later one that this is not the default value for *all* variables; it is, however, the default value for all variables we will use for most of this text.

Anonymous variables (advanced)

Although we did not explicitly declare a variable for the reference that we created when we first wrote,

```
new chapter1.Terrarium()
```

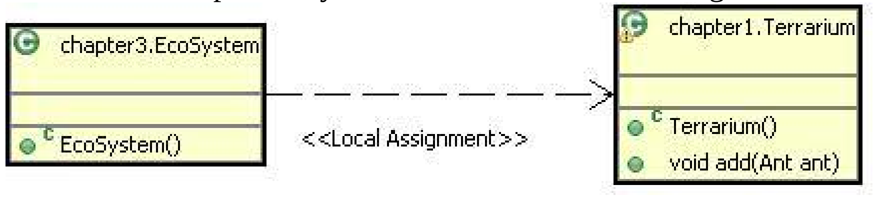
the reference had to be stored somewhere. And indeed the reference is stored in a variable, but one which is automagically created by the compiler and which does not have a name! Because the variable does not have a name, we cannot use the variable to refer to the object. A variable without a name is called an anonymous variable.

The Local Variable Dependency Relationship

The local variable dependency is very similar to the instantiation dependency, which we first saw at the end of chapter 2. There we said that the instantiation dependency exists between two classes when one of them (the "source" class) creates an instance of the other (the "target" class). In the local variable dependency relationship the source class still creates an instance of the target class, but it stores a reference to the newly created object in a local variable.

Local Variable Dependency Relationship in UML

Inside a UML diagram, two classes that are in the local variable dependency relationship have a dashed arc with an arrow head between them. The arrow points from the source class to the target class. Since both instantiation dependency and local variable dependency are variations on the same basic "dependency" relationship, the arc is the same in both cases. Green labels the arcs of these two relationships to help distinguish them in the UML diagram. The local variable dependency arc is labeled "<<Local Assignment>>":



Local Variable Dependency Relationship in Code

The relationship shown in the diagram above, if drawn in Green, will result in the following code being generated:

```
package chapter3;
```

[4] Not all languages require compilers to provide default values for uninitialized variables (variables declared but not assigned a value). The Java programming language does require this.

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

Notice that Green automatically generates a variable name, and combines the declaration of the variable with its initialization.

Relationship cardinality

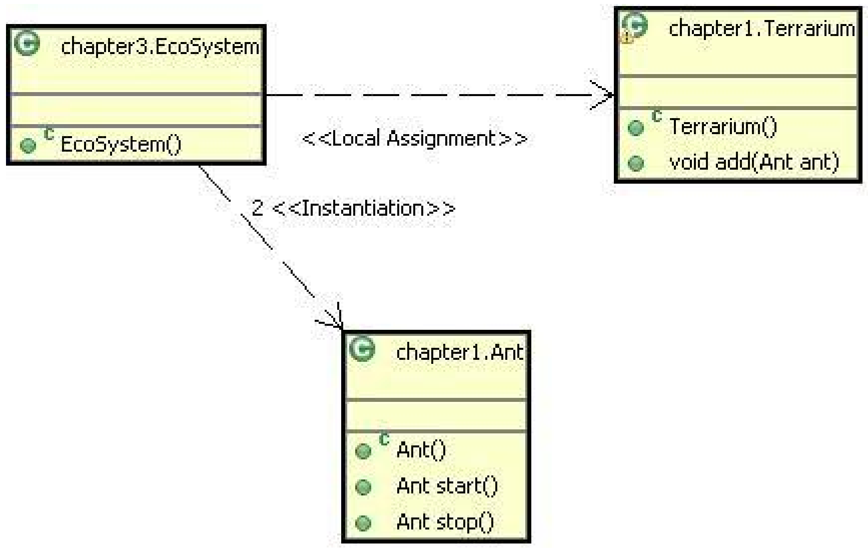
Recall how far we had gotten at the end of chapter 2 with our EcoSystem example:

```
package chapter2;

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

With what we now have learned about local variables we can create an EcoSystem (in a chapter3 package) that has one chapter1.Terrarium containing two chapter1.Ants. In this example we keep the declaration of the ourFirstTerrrarium variable separate from the initialization of the variable, which occurs in a separate statement.

```
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());
      }
}
```

The UML diagram above only showns the two classes chapter3.EcoSystem and chapter1.Terrarium. If we add chapter1.Ant into the diagram, this is what we get:



Here you can see the different labels on the two arcs. Notice also that the instantiation dependency arc is labeled with a “2” in addition to “<<Instantiation>>”. This “2” appears because the chapter3.EcoSystem class creates two instance of chapter1.Ant, and is called a cardinality annotation. When no cardinality annotation is given on a relationship line, the cardinality of the relationship is assumed to be 1.