Exploring Languages with Interpreters and Functional Programming Chapter 3

H. Conrad Cunningham

18 October 2018

Contents

3 Object-Based Paradigms 2
- 3.1 Chapter Introduction . . . 2
- 3.2 Motivation . . . 2
- 3.3 Object Model . . . 3
  - 3.3.1 Objects . . . 3
    - 3.3.1.1 Essential characteristics . . . 3
    - 3.3.1.2 Important but non-essential characteristics . . . 4
  - 3.3.2 Classes . . . 5
  - 3.3.3 Inheritance . . . 6
  - 3.3.4 Subtype polymorphism . . . 10
  - 3.3.5 Example in Python 3 . . . 12
- 3.4 Prototype-based Paradigm . . . 14
  - 3.4.1 Prototype concepts . . . 14
  - 3.4.2 Lua as object-based language . . . 15
  - 3.4.3 Example in Lua . . . 17
  - 3.4.4 Observations . . . 20
- 3.5 What Next? . . . 21
- 3.6 Exercises . . . 21
- 3.7 Acknowledgements . . . 21
- 3.8 References . . . 22
- 3.9 Terms and Concepts . . . 22

Copyright (C) 2016, 2017, 2018, H. Conrad Cunningham
Professor of Computer and Information Science
University of Mississippi
211 Weir Hall
P.O. Box 1848
University, MS 38677

(662) 915-5358

Browser Advisory: The HTML version of this textbook requires a browser that supports the display of MathML. A good choice as of October 2018 is a recent version of Firefox from Mozilla.

3 Object-Based Paradigms

3.1 Chapter Introduction

The imperative-declarative taxonomy described in the previous chapter divides programming styles and language features on how they handle state and how they are executed.

The previous chapter also mentioned other paradigms such as procedural, modular, object-based, and concurrent.

The dominant paradigm since the early 1990s has been the *object-oriented paradigm*. Because this paradigm is likely familiar with most readers, it is useful to examine it in more detail.

Thus the goals of this chapter are to examine the characteristics of:

- the object-oriented paradigm
- related paradigms such as the object-based, class-based, and prototype-based paradigms

3.2 Motivation

In contemporary practice, most software engineers approach the design of programs from an object-oriented perspective.

Key idea (notion?) in object orientation: The real world can be accurately described as a collection of objects that interact.

This approach is based on the following *assumptions*:

1. Describing large, complex systems as interacting objects make them *easier to understand* than otherwise.
2. The *behaviors* of real world objects tend to be *stable* over time.
3. The different *kinds* of real world objects tend to be *stable*. (That is, new kinds appear slowly; old kinds disappear slowly.)
4. *Changes* tend to be *localized* to a few objects.

Assumption 1 simplifies requirements analysis, software design, and implementation—makes them more reliable.

Assumptions 2 and 3 support reuse of code, prototyping, and incremental development.

Assumption 4 supports design for change.

The object-oriented approach to software development:

- uses the same basic entities (i.e. objects) throughout the software development lifecycle
- identifies the basic objects during analysis
- identifies lower-level objects during design, reusing existing object descriptions where appropriate
- implements the objects as software structures (e.g. Java classes)
- maintains the object behaviors

3.3 Object Model

We discuss object orientation in terms of an object model. Our *object model* includes four basic components:

1. objects (i.e. abstract data structures)
2. classes (i.e. abstract data types)
3. inheritance (hierarchical relationships among abstract data types)
4. subtype polymorphism

Some writers consider *dynamic binding* a basic component of object orientation. Here we consider it an implementation technique for subtype polymorphism.

Now let's consider each of four components of the object model.

3.3.1 Objects

For languages in the object-based paradigms, we require that objects exhibit three essential charactercs. Some writers consider one or two other other characteristics as essential. Here we consider these as important but non-essential characteristics of the object model.

3.3.1.1 Essential characteristics

An *object* must exhibit three *essential* characteristics:

a. state

b. operations

c. identity

An object is a separately identifiable entity that has a set of operations and a state that records the effects of the operations. An object is typically a *first-class* entity that can be stored in variables and passed to or returned from subprograms.

The *state* is the collection of information held (i.e. stored) by the object.

- It can change over time.
- It can change as the result of an operation performed on the object.
- It cannot change spontaneously.

The various components of the state are sometimes called the *attributes* of the object.

An *operation* is a procedure that takes the state of the object and zero or more arguments and changes the state and/or returns one or more values. Objects permit certain operations and not others.

If an object is *mutable*, then an operation may change the stored state so that a subsequent operation on that object acts upon the modified state; the language is thus imperative.

If an object is *immutable*, then an operation cannot change the stored state; instead the operation returns a new object with the modified state.

Identity means we can distinguish between two distinct objects (even if they have the same state and operations).

As an example, consider an object for a student desk in a simulation of a classroom.

- A student desk is distinct from the other student desks and, hence, has a unique *identity*.
- The relevant *state* might be attributes such as location, orientation, person using, items in the basket, items on top, etc.
- The relevant *operations* might be state-changing operations (called *mutator*, setter, or command operations) such as "move the desk", "seat student", or "remove from basket" or might be state-observing operations (called *accessor*, getter, observer, or query operations) such as "is occupied" or "report items on desktop".

A language is *object-based* if it supports objects as a language feature.

Object-based languages include Ada, Modula, Clu, C++, Java, Scala, C#, Smalltalk, and Python 3.

Pascal (without module extensions), Algol, Fortran, and C are not inherently object-based.

3.3.1.2 Important but non-essential characteristics

Some writers require that an object have additional characteristics, but this book considers these as important but *non-essential* characteristics of objects:

d. encapsulation

e. independent lifecycle

The state may be *encapsulated* within the object—that is, not be directly visible or accessible from outside the object.

The object may also have an *independent lifecycle*—that is, the object may exist independently from the program unit that created it. Its lifetime is not determined by the program unit that created it.

We do not include these as essential characteristics because they do not seem required by the object metaphor.

Also, some languages we wish to categorize as object-based do not exhibit one or both of these characteristics. There are languages that use a modularization feature to enforce encapsulation separately from the object (or class) feature. Also, there are languages that may have local "objects" within a function or procedure.

In languages like Python 3, Lua, and Oberon, objects exhibit an independent lifecycle but do not themselves enforce encapsulation. Encapsulation may be supported by the module mechanism (e.g. in Oberon and Lua) or partly by a naming convention (e.g. in Python 3).

In C++, some objects may be local to a function and, hence, be allocated on the runtime stack. These objects are deallocated upon exit from the function. These object may exhibit encapsulation, but do not exhibit an independent lifecycle.

3.3.2 Classes

A *class* is a template or factory for creating objects.

- A class describes a collection of related objects (i.e. *instances* of the class).
- Objects of the same class have common operations and a common set of possible states.
- The concept of class is closely related to the concept of *type*.

A class description includes definitions of:

- operations on objects of the class
- the set of possible states

As an example, again consider a simulation of a classroom. There might be a class `StudentDesk` from which specific instances can be created as needed.

An object-based language is *class-based* if the concept of class occurs as a language feature and every object has a class.

Class-based languages include Clu, C++, Java, Scala, C#, Smalltalk, Ruby, and Ada 95. Ada 83 and Modula are not class-based.

At their core, JavaScript and Lua are object-based but not class-based.

In statically typed, class-based languages such as Java, Scala, C++, and C# classes are treated as types. Instances of the same class have the same (nominal) type.

However, some dynamically typed languages may have a more general concept of type: If two objects have the same set of operations, then they have the same type regardless of how the object was created. Languages such as Smalltalk and Ruby have this characteristic—sometimes informally called *duck typing*. (If it walks like a duck and quacks like a duck, then it is a duck.)

See the Types chapter for more discussion of types.

3.3.3 Inheritance

A class C *inherits* from class P if C's objects form a subset of P's objects.

- Class C's objects must support all of the class P's operations (but perhaps are carried out in a special way).
- Class C may support additional operations and an extended state (i.e. more information fields).
- Class C is called a *subclass* or a *child* or *derived class*.
- Class P is called a *superclass* or a *parent* or *base* class.
- Class P is sometimes called a *generalization* of class C; class C is a *specialization* of class P.

The importance of inheritance is that it encourages sharing and reuse of both design information and program code. The shared state and operations can be described and implemented in base classes and shared among the subclasses.

As an example, again consider the student desks in a simulation of a classroom. The `StudentDesk` class might be derived (i.e. inherit) from a class `Desk`, which in turn might be derived from a class `Furniture`. In diagrams, there is a convention to draw arrows (e.g. ⟵) from the subclass to the superclass.

`Furniture` ⟵ `Desk` ⟵ `StudentDesk`

The simulation might also include a `ComputerDesk` class that also derives from `Desk`.

`Furniture` ⟵ `Desk` ⟵ `ComputerDesk`

We can also picture the above relationships among these classes with a class diagram as shown in Figure 3-1.

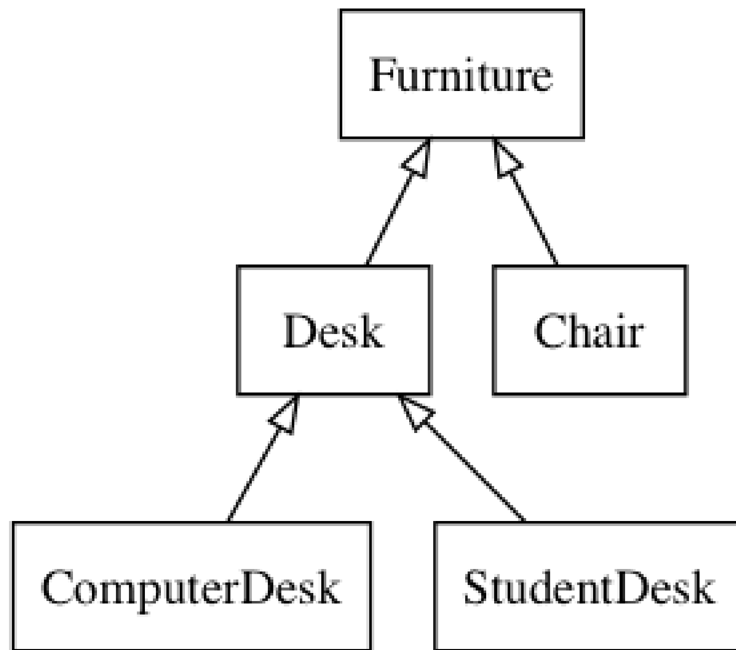

Figure 3-1: Classroom Simulation Inheritance Hierarchy

In Java and Scala, we can express the above inheritance relationships using the `extends` keyword as follows.

```
class Furniture  // extends cosmic root class for references
{   ...   }      // (java.lang.Object, scala.AnyRef)

class Desk extends Furniture
{   ...   }

class StudentDesk extends Desk
{   ...   }

class ComputerDesk extends Desk
{   ...   }
```

Both `StudentDesk` and `ComputerDesk` objects will need operations to simulate a `move` of the entity in physical space. The `move` operation can thus be implemented in the `Desk` class and shared by objects of both classes.

Invocation of operations to `move` either a `StudentDesk` or a `ComputerDesk` will be bound to the general `move` in the `Desk` class.

The `StudentDesk` class might inherit from a `Chair` class as well as the `Desk`

class.

`Furniture` ⟵ `Chair` ⟵ `StudentDesk`

Some languages support *multiple inheritance* as shown in Figure 3-2 for `StudentDesk` (e.g. C++, Eiffel, Python 3). Other languages only support a single inheritance hierarchy.

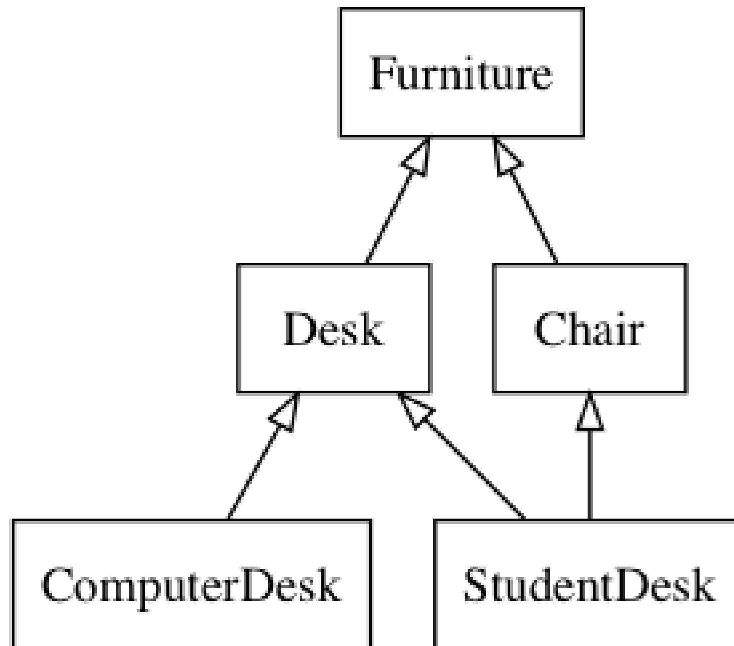

Figure 3-2: Classroom Simulation with Multiple Inheritance

Because multiple inheritance is both difficult to use correctly and to implement in a compiler, the designers of Java and Scala did not include multiple inheritance of classes as features. Java has a single inheritance hierarchy with a top-level class named `Object` from which all other classes derive (directly or indirectly). Scala is similar, with the corresponding top-level class named `AnyRef`.

```
class StudentDesk extends Desk, Chair  // NOT VALID in Java
{   ...   }
```

To see some of the problems in implementing multiple inheritance, consider the above example. Class `StudentDesk` inherits from class `Furniture` through two different paths. Do the data fields of the class `Furniture` occur once or twice? What happens if the intermediate classes `Desk` and `Chair` have conflicting definitions for a data field or operation with the same name?

The difficulties with multiple inheritance are greatly decreased if we restrict ourselves to inheritance of class *interfaces* (i.e. the signatures of a set of operations)

rather than a supporting the inheritance of the class *implementations* (i.e. the instance data fields and operation implementations). Since interface inheritance can be very useful in design and programming, the Java designers introduced a separate mechanism for that type of inheritance.

The Java `interface` construct can be used to define an interface for classes separately from the classes themselves. A Java `interface` may inherit from (i.e. `extend`) zero or more other `interface` definitions.

```
interface Location3D
{   ...   }

interface HumanHolder
{   ...   }

interface Seat extends Location3D, HumanHolder
{   ...   }
```

A Java `class` may inherit from (i.e. `implement`) zero or more interfaces as well as inherit from (i.e. `extend`) exactly one other `class`.

```
interface BookHolder
{   ...   }

interface BookBasket extends Location3D, BookHolder
{   ...   }

class StudentDesk extends Desk implements Seat, BookBasket
{   ...   }
```

Figure 3-3 shows this interface-based inheritance hierarchy for the classroom simulation example. The dashed lines represent the `implements` relationship.

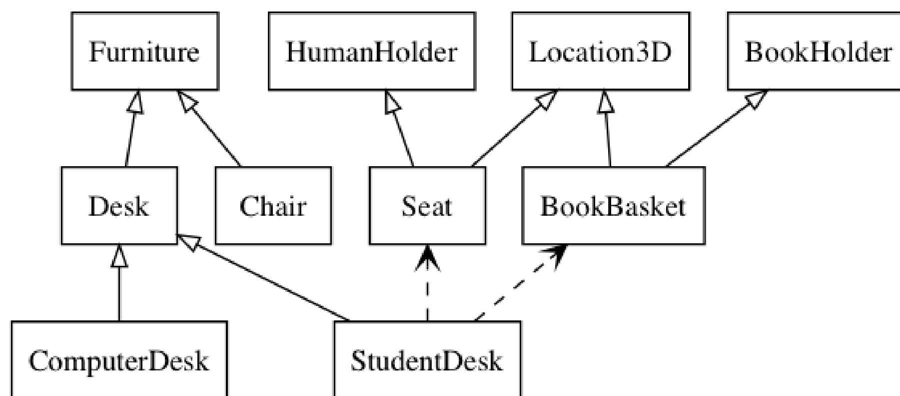

Figure 3-3: Classroom Simulation with Interfaces

This definition requires the `StudentDesk` class to provide actual implementations for all the operations from the `Location3D`, `HumanHolder`, `BookHolder`, `Seat`, and `BookBasket` interfaces. The `Location3D` operations will, of course, need to be implemented in such a way that they make sense as part of both the `HumanHolder` and `BookHolder` abstractions.

The Scala `trait` provides a more powerful, and more complex, mechanism than Java's original `interface`. In addition to signatures, a `trait` can define method implementations and data fields. These traits can be added to a class in a controlled, linearized manner to avoid the semantic and implementation problems associated with multiple inheritance of classes. This is called *mixin* inheritance.

Java 8+ generalizes interfaces to allow default implementations of methods.

Most statically typed languages treat subclasses as *subtypes*. That is, if `C` is a subclass of `P`, then the objects of type `C` are also of type `P`. We can *substitute* a `C` object for a `P` object in all cases.

However, the inheritance mechanism in languages in most class-based languages (e.g. Java) does not automatically preserve substitutability. For example, a subclass can change an operation in the subclass to do something totally different from the corresponding operation in the parent class.

3.3.4 Subtype polymorphism

The concept of *polymorphism* (literally "many forms") means the ability to hide different implementations behind a common interface. Polymorphism appears in several forms in programming languages. We will discuss these more later.

Subtype polymorphism (sometimes called *polymorphism by inheritance*, *inclusion polymorphism*, or *subtyping*) means the association of an operation invocation (i.e. procedure or function call) with the appropriate operation implementation in an inheritance (subtype) hierarchy.

This form of polymorphism is usually carried out at run time. That implementation is called *dynamic binding*. Given an object (i.e. class instance) to which an operation is applied, the system will first search for an implementation of the operation associated with the object's class. If no implementation is found in that class, the system will check the superclass, and so forth up the hierarchy until an appropriate implementation is found. Implementations of the operation may appear at several levels of the hierarchy.

The combination of dynamic binding with a well-chosen inheritance hierarchy allows the possibility of an instance of one subclass being substituted for an instance of a different subclass during execution. Of course, this can only be done when none of the extended operations of the subclass are being used.

As an example, again consider the simulation of a classroom. As in our discussion of inheritance, suppose that the `StudentDesk` and `ComputerDesk` classes are derived from the `Desk` class and that a general `move` operation is implemented as a part of the `Desk` class. This could be expressed in Java as follows:

```
class Desk extends Furniture
{   ...
    public void move(...)
    ...
}

class StudentDesk extends Desk
{   ...
    // no move(...) operation here
    ...
}

class ComputerDesk extends Desk
{   ...
    // no move(...) operation here
    ...
}
```

As we noted before, invocation of operations to `move` either a `StudentDesk` or a `ComputerDesk` instance will be bound to the general `move` in the `Desk` class.

Extending the example, suppose that we need a special version of the `move` operation for `ComputerDesk` objects. For instance, we need to make sure that the computer is shut down and the power is disconnected before the entity is moved.

To do this, we can define this special version of the `move` operation and associate it with the `ComputerDesk` class. Now a call to `move` a `ComputerDesk` object will be bound to the special `move` operation, but a call to `move` a `StudentDesk` object will still be bound to the general `move` operation in the `Desk` class.

The definition of `move` in the `ComputerDesk` class is said to *override* the definition in the `Desk` class.

In Java, this can be expressed as follows:

```
class Desk extends Furniture
{   ...
    public void move(...)
    ...
}

class StudentDesk extends Desk
{   ...
```

```
        // no move(...) operation here
        ...
    }

    class ComputerDesk extends Desk
    {   ...
        public void move(...)
        ...
    }
```

A class-based language is *object-oriented* if class hierarchies can be incrementally defined by an inheritance mechanism and the language supports polymorphism by inheritance along these class hierarchies.

Object-oriented languages include C++, Java, Scala, C#, Smalltalk, and Ada 95. The language Clu is class-based, but it does not include an inheritance facility.

Other object-oriented languages include Objective C, Object Pascal, Eiffel, and Oberon 2.

3.3.5 Example in Python 3

Python 3 is a dynamically typed language with support for imperative, procedural, modular, object-oriented, and (to a limited extent) functional programming styles. It's object model supports state, operations, identity, and an independent lifecycle. It provides some support for encapsulation. It has classes, single and multiple inheritance, and subtype polymorphism.

Let's again examine the counting problem from Chapter 2 from the standpoint of object-oriented programming in Python 3. The following code defines a class named `CountingOO`. It defines four instance methods and two instance variables.

Note: By *instance variable* and *instance method* we mean variables and instances associated with an object, an instance of a class.

```
class CountingOO:                   # (1)

    def __init__(self,c,m):         # (2,3)
        self.count = c              # (4)
        self.maxc  = m

    def has_more(self,c,m):         # (5)
        return c <= m

    def adv(self):                  # (6)
        self.count = self.count + 1

    def counter(self):              # (7)
```

```
        while self.has_more(self.count,self.maxc):
            print(f'{self.count}') # (8)
            self.adv()
```

The following notes explain the numbered items in the above code.

1. By default, a Python 3 class inherits from the cosmic root class `object`. If a class inherits from some other class, then we place the parent class's name in parenthesis after the class name, as with class `Times2` below. (Python 3 supports multiple inheritance, so there can be multiple class names separated by commas.)

2. Python 3 classes do not normally have explicit constructors, but we often define an initialization method which has the special name `__init__`.

3. Unlike object-oriented languages such as Java, Python 3 requires that the receiver object be passed explicitly as the first parameter of instance methods. By convention, this is a parameter named `self`.

4. An instance of the class `CountingOO` has two instance variables, `count` and `maxc`. Typically, we create these dynamically by explicitly assigning a value to the name. We can access these values in expressions (e.g. `self.count`).

5. Method `has_more()` is a function that takes the receiver object and values for the current count and maximum values and returns `True` if and only there are additional values to generate. (Although an instance method, it does not access the instance's state.)

6. Method `adv()` is a procedure that accesses and modifies the state (i.e. the instance variables), setting `self.count` to a new value closer to the maximum value `self.maxc`.

7. Method `counter()` is a procedure intended as the primary public interface to an instance of the class. It uses function method `has_more()` to determine when to stop the iteration, procedure method `adv()` to advance the variable `count` from one value to the next value, and the `print` function to display the value on the standard output device.

8. Expression `f'{self.count}'` is a Python 3.7 interpolated string.

In terms of the *Template Method* design pattern [Gamma 1995], `counter` is intended as a *template method* that encodes the primary algorithm and is not intended to be overridden. Methods `has_more()` and `adv()` are intended as *hook methods* that are often overriden to give different behaviors to the class.

Consider the following fragment of code.

```
ctr = CountingOO(0,10)
ctr.counter()
```

The first line above creates an instance of the `CountingOO` class, initializes its instance variables `count` and `maxc` to 0 and 10, and stores the referene in variable

`ctr`. The call `ctr.counter()` thus prints the values 0 to 10, one per line, as do the programs from Chapter 2.

However, we can create a subclass that overrides the definitions of the hook methods `has_more()` and `adv()` to give quite different behavior without modifying class `CountingOO`.

```
class Times2(CountingOO):    # inherits from CountingOO

    def has_more(self,c,m):  # overrides
        return c != 0 and abs(c) <= abs(m)

    def adv(self):           # overrides
        self.count = self.count * 2
```

Now consider the following code fragment.

```
ctr2 = Times2(-1,10)
ctr2.counter()
```

This generates the sequence of values -1, -2, -4, and -8, printed one per line.

The call to any method on an instance of class `Times2` is polymorphic. The system dynamically searches up the class hierarchy from `Times2` to find the appropriate function. It finds `has_more()` and `adv()` in `Times2` and `counter()` in parent class `CountingOO`.

The code for this section is in source file `CountingOO.py`.

3.4 Prototype-based Paradigm

Classes and inheritance are not the only way to support relationships among objects in object-based languages. Another approach of growing importance is the use of *prototypes*.

3.4.1 Prototype concepts

A *prototype-based* language does not have the concept of class as defined above. It just has objects. Instead of using a class to instantiate a new object, a program copies (or clones) an existing object—the *prototype*—and modifies the copy to have the needed attributes and operations.

Each prototype consists of a collection of *slots*. Each slot is filled with either a data attribute or an operation.

This cloning approach is more flexible than the class-based approach.

In a class-based language, we need to define a new class or subclass to create a variation of an existing type. For example, we may have a `Student` class. If we

want to have students who play chess, then we would need to create a new class, say `ChessPlayingStudent`, to add the needed data attributes and operations.

Aside: Should `Student` be the parent `ChessPlayingStudent`? or should `ChessPlayer` be the parent? Or should we have fields of `ChessPlayingStudent` that hold `Student` and `ChessPlayer` objects?

In a class-based language, the boundaries among categories of objects specified by classes should be *crisply defined*. That is, an object is in a particular class or it is not. Sometimes this crispness may be unnatural.

In a prototype-based language, we simply clone a student object and add new slots for the added data and operations. This new object can be a prototype for further objects.

In a prototype-based language, the boundaries between categories of objects created by cloning may be fuzzy. One category of objects may tend to blend into others. Sometimes this fuzziness may be more natural.

Consider categories of people associated with a university. These categories may include `Faculty`, `Staff`, `Student`, and `Alumnus`. Consider a *student* who gets a BSCS degree, then accepts a *staff* position as a programmer and stays a student by starting an MS program part-time, and then later teaches a course as a graduate student. The same person who started as a student thus evolves into someone who is in several categories later. And he or she may also be a chess player.

Instead of static, class-based inheritance and polymorphism, some languages exhibit prototype-based *delegation*. If the appropriate operation cannot be found on the current object, the operation can be delegated to its prototype, or perhaps to some other related, object. This allows dynamic relationships along several dimensions. It also means that the "copying" or "cloning" may be partly logical rather than physical.

Prototypes and delegation are more basic mechanisms than inheritance and polymorphism. The latter can often be implemented (or perhaps "simulated") using the former.

Self, NewtonScript, JavaScript, Lua, and Io are prototype-based languages. Python with package prototype.py is also prototype-based.

Let's look at Lua as a prototype-based language.

3.4.2 Lua as object-based language

Lua is a dynamically typed, multiparadigm language. The language designers stress the following design principles [Lua 2018]:

- portability
- embeddability

- efficiency
- simplicity

To realize these principles, the core language implementation:

- can only use standard C and the standard C library
- must be efficient in use of memory and processor time (i.e. keep the interpreter small and fast)
- must support interoperability with C programs in both directions (i.e. can call or be called by C programs)

C is ubiquitous, likely being the first higher-level language implemented for any new machine, whether a small microcontroller or a large multiprocessor. So this implementation approach supports the portability, embeddability, and efficiency design goals.

Because of Lua's strict adherence to the above design principles, it has become a popular language for extending other applications with user-written scripts or templates. For example, it is used for this purpose in some computer games and by Wikipedia. Also, Pandoc, the document conversion tool used in production of this textbook, enables scripts to be written in Lua. (The Pandoc program itself is written in Haskell.)

The desire for a simple but powerful language led the designers to adopt an approach that *separates mechanisms from policy*. As noted on the Lua website [Lua 2018],

> A fundamental concept in the design of Lua is to provide meta-mechanisms for implementing features, instead of providing a host of features directly in the language. For example, although Lua is not a pure object-oriented language, it does provide meta-mechanisms for implementing classes and inheritance. Lua's meta-mechanisms bring an economy of concepts and keep the language small, while allowing the semantics to be extended in unconventional ways.

Lua provides a small set of quite powerful primitives. For example, it includes only one data structure—the *table* (dictionary, map, or object in other languages)—but ensures that it is efficient and flexible for a wide range of uses.

Lua's tables are *objects* as described earlier in this chapter. Each object has its own:

- *state* (i.e. values associated with keys)
- *identity* independent of state
- *lifecycle* independent of the code that created it

In addition, a table can have its own *operations* by associating function closures with keys.

Note: By *function closure*, we mean the function's definition plus aspects of its environment necessary (e.g. variables variables outside the function) necessary for the function to be executed.

So a key in the table represents a *slot* in the object. The slot can be occupied by either a data attribute's value or the function closure associated with an operation.

Lua tables do not directly support *encapsulation*, but there are ways to build structures that encapsulate key data or operations.

Lua's *metatable* mechanism, particularly the `__index` *metamethod*, enables an access to an undefined key to be delegated to another table (or to result in a call of a specified function).

Thus tables and metatables enable the prototype-based paradigm as illustrated in the next section.

As in Python 3, Lua requires that the *receiver* object be passed as an argument to object-based function and procedure calls. By convention, it is passed as the first argument, as shown below.

```
obj.method(obj, other_arguments)
```

Lua has a bit of *syntactic sugar*—the : operator—to make this more convenient. The following Lua expression is equivalent to the above.

```
obj:method(other_arguments)
```

The Lua interpreter evaluates the expression `obj` to get the receiver object (i.e. table), then retrieves the function closure associated with the key named `method` from the receiver object, then calls the function, passing the receiver object as its first parameter. In the body of the function definition, the receiver object can be referenced by parameter name `self`.

We can use a similar notation to define functions to be methods associated with objects (tables).

3.4.3 Example in Lua

The Lua code below, from file `CountingPB.lua`, implements a Lua module similar to the Python 3 `CountingOO` class given in an earlier section. It illustrates how to define Lua modules as well as prototypes.

```
-- File CountingPB.lua
local CountingPB = {count = 1, maxc = 0} -- (1)

function CountingPB:new(mixin)            -- (2)
   mixin = mixin or {}                    -- (5)
   local obj = { __index = self }         -- (4)
```

```
   for k, v in pairs(mixin) do            -- (5)
      if k ~= "__index" then
         obj[k] = v
      end
   end
   return setmetatable(obj,obj)           -- (6,7)
end

function CountingPB:has_more(c,m)         -- (2)
   return c <= m
end

function CountingPB:adv()                 -- (2)
   self.count = self.count + 1
end

function CountingPB:counter()             -- (2)
   while self:has_more(self.count,self.maxc) do
      print(self.count)
      self:adv()
    end
end

return CountingPB                         -- (3)
```

The following notes explain the numbered steps in the above code.

1. Create module object `CountingPB` as a Lua table with default values for data attributes `count` and `maxc`. This object is also the top-level prototype object.

2. Define methods (i.e. functions) `new()`, `has_more()`, `adv()`, and `counter()` and add them to the `CountingPB` table. The key is the function's name and the value is the function's closure.

   Method `new()` is the constructor for clones.

3. Return `CountingPB` when the module file `CountingPB.lua` is imported with a `require` call in another Lua module or script file.

Method `new` is what constructs the clones. This method:

4. Creates the clone initially as a table with only the `__index` set to the object that called `new` (i.e. the receiver object `self`).

5. Copies the method `new`'s parameter `mixin`'s table entries into the clone. This enables existing data and method attributes of the receiver object `self` to be redefined and new data and method attributes to be added to the clone.

If parameter `mixin` is undefined or an empty table, then no changes are made to the clone.

6. Sets the clone's metatable to be the clone's table itself. In step 4, we had set its metamethod `__index` to be the receiver object `self`.
7. Returns the clone object (a table) as is the convention for Lua modules.

If a Lua program accesses an undefined key of a table (or object), then the interpreter checks to see whether the table has a metatable defined.

- If no metatable is set, then the result of the access is a `nil` (meaning undefined).
- If a metatable is set, then the interpreter uses the `__index` metamethod to determine what to do. If `__index` is a table, then the access is delegated to that table. If `__index` is set a function closure, then the interpreter calls that function. If there is no `__index`, then it returns a `nil`.

We can load the `CountingPB.lua` module as follows:

```
local CountingPB = require "CountingPB"
```

Now consider the Lua assignment below:

```
x = CountingPB:new({count = 0, maxc = 10})
```

This creates a clone of object `CountingPB` and stores it in variable `x`. This clone has its own data attributes `count` and `maxc`, but it delegates method calls back to object `CountingPB`.

If we execute the call `x:counter()`, we get the following output:

```
0
1
2
3
4
5
6
7
8
9
10
```

Now consider the Lua assignment:

```
y = x:new({count = 10, maxc = 15})
```

This creates a clone of object in `x` and stores the clone in variable `y`. The `y` object has different values for `count` and `maxc`, but it delegates the method calls to `x`, which, in turn, delegates them on to `CountingPB`.

If we execute the call `y:counter()`, we get the following output:

```
10
11
12
13
14
15
```

Now, consider the following Lua assignment:

```
z = y:new( { maxc = 400,
             has_more = function (self,c,m)
                return c ~= 0 and math.abs(c) <= math.abs(m)
             end,
             adv = function(self)
                self.count = self.count * 2
             end,
             bye = function(self) print(self.msg) end
             msg = "Good-Bye!" } )
```

This creates a clone of object `y` that keeps `x`'s current value of `count` (which is 16 after executing `y:counter()`), sets a new value of `maxc`, overrides the definitions of methods `has_more()` and `adv()`, and defines new method `bye()` and new data attribute `msg`.

If we execute the call `z:counter()` followed by `z:bye()`, we get the following output:

```
16
32
64
128
256
Good-Bye!
```

The source code for this example is in file `CountingPB.lua`. The example calls are in file `CountingPB_Test.lua`.

3.4.4 Observations

How does the prototype-based (PB) paradigm compare with the object-oriented (OO) paradigm?

- The OO paradigm as implemented in a language usually enforces a particular discipline or policy and provides syntactic and semantic support for that policy. However, it makes programming outside the policy difficult.
- The PB paradigm is more flexible. It provides lower-level mechanisms and little or no direct support for a particular discipline or policy. It allows programmers to define their own policies, simple or complex policies

depending on the needs. These policies can be implemented in libraries and reused. However, PB can result in different programmers or different software development shops using incompatible approaches.

Whatever paradigm we use (OO, PB, procedural, functional, etc.), we should be careful and be consistent in how we design and implement programs.

3.5 What Next?

In this and the previous chapter, we explored various programming paradigms.

In the next chapter, we begin examining Haskell, looking at our first simple programs and how to execute those programs with the interactive interpreter.

In subsequent chapters, we look more closely at the concepts of type introduced in this chapter and abstraction introduced in the previous chapter.

3.6 Exercises

1. This chapter used Python 3 to illustrate the object-oriented paradigm. Choose a language such as Java, C++, or C#. Describe how it can be used to write programs in the object-oriented paradigm. Show the `CountingOO` example in the chosen language.

2. C is a primarily procedural language. Describe how C can be used to implement object-based programs. Show the `CountingOO` example in the chosen language.

3.7 Acknowledgements

In Summer and Fall 2016, I adapted and revised much of this work from my previous materials:

- Object-Oriented programming paradigm from my notes *Introduction to Object Orientation* [Cunningham 2014], which I wrote originally for the first C++ (CSci 490) and Java-based (CSci 211) classes at UM in 1996 but expanded and adapted for other courses; these notes were influenced by Horstmann [Horstmann 1995], Budd [Budd 2000], and other sources

- Prototype-based programming paradigm from draft notes on that topic [Cunningham 2016]; these notes were influenced by [Craig 2007], [Ierusalimschy 2016], and other sources

In 2017, I continued to develop this material as a part of Chapter 1, Fundamentals, of my 2017 Haskell-based programming languages textbook.

In 2018 I reorganized and expanded the previous Fundamentals chapter into four chapters for the 2018 version of the textbook, now titled *Exploring Languages with Interpreters and Functional Programming*. These are Chapter 1, Evolution of Programming Languages; Chapter 2, Programming Paradigms; Chapter 3, Object-Based Paradigms (this chapter); and Chapter 80, Review of Relevant Mathematics.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

3.8 References

[**Budd 2000**]**:** Timothy Budd. *Understanding Object-Oriented Programming with Java*, Updated Edition, Addison Wesley, 2000.

[**Craig 2007**]**:** Iain D. Craig. *Object-Oriented Programming Languages*, Springer 2007. (Especially chapter 1 "Introduction" and chapter 3 "Prototype and Actor Languages",)

[**Cunningham 2014**]**:** H. Conrad Cunningham. *Introduction to Object Orientation*, 1996-2014.

[**Cunningham 2016**]**:** H. Conrad Cunningham. *Prototype-Based Programming Paradigm*, 2016.

[**Gamma 1995**]**:** Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995.

[**Horstmann 1995**]**:** Cay S. Horstmann. *Mastering Object-Oriented Design in C++*, Wiley, 1995. (Especially chapters 3-6 on "Implementing Classes", "Interfaces", "Object-Oriented Design", and "Invariants" which influenced my views on object-oriented design and programming)

[**Ierusalimschy 2016**]**:** Roberto Ierusalimschy. *Programming in Lua*, Fourth Edition, Lua.org, 2013; Third Edition, 2013.

[**Lua 2018**]**:** LabLua. *Lua: The Programming Language* website, PUC-Rio, 2018. <http://www.lua.org/about.html:, accessed 23 August 2018.

3.9 Terms and Concepts

Object (state, operations, identity, encapsulation, independent lifecycle, mutable, immutable), object-based language, class, type, class-based language, inheritance, subtype, interface, polymorphism, subtype polymorphism (subtyping, inclusion polymorphism, polymorphism by inheritance), dynamic binding, object-oriented language, prototype, clone, slot, delegation, prototype-based language, embeddability, Lua tables, metatables, and metamethods, function closure.