Exploring Languages with Interpreters and Functional Programming

Chapter 3

H. Conrad Cunningham

19 August 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 ................... 9
  3.3.5 Example in Python 3 .................... 10
3.4 Prototype-based Paradigm .................... 13
3.5 What Next? .................................. 14
3.6 Exercises .................................... 14
3.7 Acknowledgements ........................... 14
3.8 References .................................. 15
3.9 Terms and Concepts ......................... 15

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 August 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)
• maintain 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 characteristics. 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.

- The student desks are distinct from each other 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 possible set of 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, it is the 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

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

```java
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 above for StudentDesk (e.g. C++, Eiffel, Python 3). Other languages only support a single inheritance hierarchy.

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
{
   ...
}

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:

```java
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
A computer desk instance will be bound to the general move operation 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:

```java
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.

```python
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 have explicit constructors, but we often define an initialization method with 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 \texttt{adv()} is a procedure that accesses and modifies the state (i.e. the
instance variables), setting \texttt{self.count} to a new value closer to the maxi-
mum value \texttt{self.maxc}.

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

In terms of the \textit{Template Method} design pattern [Gamma 1995], \texttt{counter}
is intended as a \textit{template method} that encodes the primary algorithm and is not
intended to be overridden. Methods \texttt{has\_more()} and \texttt{adv()} are intended as
\textit{hook methods} which are often overriden to give different behaviors to the class.

Consider the following fragment of code.

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

The first line above creates an instance of the \texttt{CountingOO} class, initializes its
instance variables \texttt{count} and \texttt{maxc} to 0 and 10, and stores the referene in variable\texttt{ctr}. The call \texttt{ctr \textunderscore 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 meth-
ods \texttt{has\_more()} and \texttt{adv()} to give quite different behavior without modifying
class \texttt{CountingOO}.

```python
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.

```python
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 \texttt{Times2} is polymorphic. The
system dynamically searches up the class hierarchy from \texttt{Times2} to find the
appropriate function. It finds \texttt{has\_more()} and \texttt{adv()} in \texttt{Times2} and \texttt{counter()} in
parent class \texttt{CountingOO}.
3.4 Prototype-based Paradigm

TODO: Give example

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.

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.

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, JavaScript, and Lua are prototype-based languages.

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 2014b], 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

- Prototype-based programming paradigm from draft notes on that topic [Cunningham 2016a]

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

In Spring and Summer 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


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


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