Chapter 2 Data Abstraction

Objectives

- To learn how abstract data types are used in software design.
- To review the basic principles and techniques of object-oriented design.
- To learn about unit testing and how to write unit tests in Python.
- To learn about operator overloading and how to overload operators in Python.

2.1 Overview

Algorithms are one fundamental building block of programs. In Chapter 1, we saw the benefits that come from separating the idea of what a function does from the details of how it is implemented. In this chapter, we'll take a look at the data that our programs process. Separating behavior from implementation is even more powerful when we consider data objects. This process of *data abstraction* is a foundational concept that must be mastered in order to build practical software systems. Computer scientists formalize the idea of data abstraction in terms of *abstract data types* (ADTs). Abstract data types, in turn, are the foundation for object-oriented programming, which is the dominant development method for large systems.

We'll start out by examining ADTs and how they relate to object-oriented programming. Along the way we'll show how object-oriented programming can be used to extend a programming language with new data types that can make it more suitable for solving problems in new domains. In languages that support a special technique known as *operator overloading*, new data types can be made to look and act just like the language's own built-in types.

Using ADTs and objects, program design becomes a process of breaking a problem into smaller pieces: a set of cooperating objects that provide most of the program's functionality. As these smaller pieces are implemented, they can be tested in isolation so that developers have confidence in their correctness before the parts are combined into a larger system. Learning how to do effective testing is another important piece of the software development puzzle.

2.2 Abstract Data Types

One important property of any value stored in a computer is its *data type*. The type of an object determines both what values it can have and what we can do with it (i.e., what operations it supports). For example, on a 32-bit computer the built-in type `int` can represent integers in the range from -2^{31} to $2^{31} - 1$ and can be used with operations such as addition (+), subtraction (-), multiplication (*) and division(/). Knowing this information, you can write programs that use `int`s without having to know how such numbers are actually stored on the computer. Using our terminology from last chapter, we would say that a program that manipulates `int` values is a client of the `int` data type.

Of course, in order for a data type to actually be useful, there must be some underlying implementation of that type. The implementation consists of both a way to represent all the possible values of the type and a set of functions that manipulate the underlying representation. Consider again the `int` data type. It is typically stored on today's computers as a 32-bit binary number. Algorithms for operations such as addition and subtraction are defined in the underlying machine hardware, and functions for input and output of `int`s are built into most programming languages.

2.2.1 From Data Type to ADT

Applying the idea of abstraction, we can separate the concerns of how data is represented from how it is used. That is, we can provide a specification for a data type that is independent of any actual implementation. Such a specification describes abstract data type . A precise and complete description allows client programs to be written without worrying about how an ADT is realized in the computer. In this way, data abstraction extends the advantages of implementation independence. We can delay decisions about how data should be represented in our programs until we have sufficient information about how that data is going to be used. We can also go in and change a representation, and the abstraction barrier ensures that the rest of the program will not be adversely affected.

Data abstraction is particularly important for those parts of a program that are likely to change. Major design decisions can be encapsulated in ADTs, and the implementation of the ADTs can be adjusted as necessary without affecting the rest of the program. As you will see, it is often the case that changing how data is represented can have a major impact on the efficiency of the associated operations; so, having the freedom to modify representations is a big win when trying to tune a program's efficiency.

Another advantage of ADTs is that they promote reuse. Once a relevant abstraction has been implemented, it can be used by many different client programs. Those clients are freed from the hassle of having to reinvent the data type. This allows programmers to extend programming languages with new data objects that are useful in their particular area of programming. After the ADT has been thoroughly tested, it can be used with confidence and the implementation details never have to be revisited.

2.2.2 Defining an ADT

You can think of an ADT as a collection of functions or methods that manipulate an underlying representation. The representation is really just some collection of data. To specify an ADT we just describe what the operations supported by the ADT do. We can apply the same techniques we used for specifying functions. The only difference is that a single ADT is described by a *collection* of functions.

Let's look at a simple example. Suppose we are writing some programs dealing with card games, say bridge or Texas hold 'em. A playing card could be modeled as a simple ADT. Here's a description of the ADT:

```
ADT Card:
    A simple playing card. A Card is characterized by two components:
    rank: an integer value in the range 1-13, inclusive (Ace-King)
    suit: a character in 'cdhs' for clubs, diamonds, hearts, and
          spades.

Operations:

    create(rank, suit):
        Create a new Card
        pre: rank in range(1,14) and suit in 'cdhs'
        post: returns a Card of the given rank and suit

    suit():
        Card suit
        post: Returns Card's suit as a single character
```

```
rank():
    Card rank
    post: Returns Card's rank as an int

suitName():
    Card suit name
    post: Returns one of ('clubs', 'diamonds', 'hearts',
          'spades') corrresponding to Card's suit.

rankName():
    Card rank name
    post: Returns one of ('ace', 'two', 'three', ..., 'king')
          corresponding to Card's rank.

toString():
    String representation of Card
    post: Returns string naming the Card, e.g. 'Ace of Spades'
```

Notice how this specification describes a Card in terms of some abstract attributes (rank and suit) and the things that we can do with a card. It does not describe how a Card is actually represented or how the operations are achieved. In fact, the specification doesn't even explicitly refer to any card object or parameter; it is implicit that these are the operations that can *somehow* be applied to any card.

In the process of designing an ADT, our goal is to include a complete set of operations necessary to make the ADT useful. Of course, there are many different design choices that could be made for the Card ADT. For example, we could have different names for the operations; some designers prefer to use names starting with "get" for accessing components of an ADT. Thus, they might use getSuit and getRank in place of suit and rank. Other designers might choose different types for the parameters of the various operations. Perhaps suits might be represented with ints instead of strings. Another approach is to "hide" the exact representation of suits and ranks by simply providing a set of variables representing the suits and ranks. For example, an identifier named CLUBS might be assigned to some value representing that suit, similar to the way the identifier None refers to Python's special None object. The ranks could be represented using names like ACE, TWO, THREE, etc.

As you gain experience working with ADTs, you will develop your own design sense. The most important thing to keep in mind is implementation independence. An ADT describes only a set of operations, not how those operations are implemented. One good way of "testing" the design for an ADT is to try writing some client algorithms that use it. For example, here is an algorithm that prints out the rank, suit, and "name" of all the cards in a standard deck:

```
for s in 'cdhs':
    for r in range(1,14):
        card = create(r, s)
        print 'Suit:', suit(card)
        print 'Rank:', rank(card)
        print toString(card)
```

Notice how this algorithm is expressed using a Python-like syntax, but makes use of the abstract functions of the ADT presented above. The algorithm shows us that our set of operations would be sufficient to create and print out all 52 possible cards.

2.2.3 Implementing an ADT

It is possible to design and reason about ADTs in a language-independent fashion, but once we get down to the point of implementing and using an ADT in a program, we need to fill in some details that are specific to the particular programming environment. There are numerous ways that a programmer could go about translating an ADT into a particular programming language. Virtually all languages provide the ability to define new functions, so one way of implementing an ADT is simply to write an appropriate set of functions. For example, in Python we could write a function for each `Card` operation and place them together in a module file.

Of course, in writing the functions we will need to decide how a `Card` will be represented on the computer. The abstract type has components for rank and suit. In Python, a simple representation would be to package the rank and suit together as a pair of values in a tuple. A Python tuple is an immutable (unchangeable) sequence of values. A tuple literal is indicated by enclosing a comma-separated sequence in parentheses. Using tuples, the ace of clubs would be represented by the tuple `(1,'c')` and the king of spades would be `(13,'s')`.

The underlying representation of an ADT is called the *concrete representation.* We would say that the tuple `(5,'d')` is the concrete representation of the abstract `Card` known as the five of diamonds.

Now that we have a representation for our `Card` ADT, writing the implementation code is straightforward. Here is one version:

```
# cardADT.py
#    Module file implementing the card ADT with functions

_SUITS = 'cdhs'
_SUIT_NAMES = ['clubs', 'diamonds', 'hearts', 'spades']
```

```
_RANKS = range(1,14)
_RANK_NAMES = ['Ace', 'Two', 'Three', 'Four', 'Five', 'Six',
               'Seven', 'Eight', 'Nine', 'Ten',
               'Jack', 'Queen', 'King']

def create(rank, suit):
   assert rank in _RANKS and suit in _SUITS
   return (rank,suit)

def rank(card):
    return card[0]

def suit(card):
    return card[1]

def suitName(card):
    index = _SUITS.index(suit(card))
    return _SUIT_NAMES[index]

def rankName(card):
    index = _RANKS.index(rank(card))
    return _RANK_NAMES[index]

def toString(card):
    return rankName(card) + ' of ' + suitName(card)
```

Take a look at the `create` function. It uses an `assert` to check that the preconditions for creating a card are met, and then it simply returns a rank-suit tuple. In this way, the function returns a single value that represents all the information about a particular card.

The `rank` and `suit` operations simply unpackage the appropriate part of the card tuple. Tuple components are accessed through indexing, so `card[0]` gives the first component, which is the rank, and `card[1]` gives the suit. These two operations are so simple, you might even wonder if they are necessary. Couldn't a client using the `Card` ADT simply access the suit directly by doing something like `myCard[1]`? The answer is that the client *could* do this, but it *shouldn't*. The whole point of an ADT is to uncouple the client from the implementation. If the client accesses the representation directly, then changing the representation later will break the client code. Remember this rule: clients may use an ADT only through the provided operations.

One other point worth noting about this code is the use of some special values: `_RANKS`, `_SUITS`, `_RANK_NAMES`, and `_SUIT_NAMES`. The `suitName` and `rankName` methods could have been written as large multi-way `if` statements. Instead, we have employed a *table-driven* approach. We use the `index` method to find the position of a rank or suit, and then use it to look up the corresponding name. This shortens the code and makes it much easier to modify. For example, we could easily add a fifth suit by simply adding another item to the end of _SUITS and _SUIT_NAMES.

Just in case you were wondering, there's a reason for the funny-looking variable names used for the lookup tables. The use of uppercase is a programming convention often employed for constants, that is, things that are assigned once and never changed. The leading underscore is a Python convention indicating that these names are "private" to the module. If the client imports the module via

```
from cardADT import *
```

the identifiers beginning with an underscore are not imported into the local program. This keeps implementation details, such as the use of lookup tables, from cluttering up the client's namespace (the set of defined identifiers).

Now that we have the Card ADT implementation, we can actually code up our program that prints out cards using this card module.

```
# test_cardADT.py
import cardADT

def printAll():
    for suit in 'cdhs':
        for rank in range(1,14):
            myCard = cardADT.create(rank, suit)
            print cardADT.toString(myCard)

if __name__ == '__main__':
    printAll()
```

To summarize, one way of implementing an ADT is to choose a concrete representation and then write a set of functions that manipulate that representation. If our implementation language includes modules (a la Python), we can place the implementation in a separate module so that it has its own independent namespace.

If the implementation language does not support the idea of separate modules, then we could run into trouble with the names of operations between ADTs "clashing." For example, if we were writing a program to play a card game, we might also have a deckADT representing a deck of cards. Of course, the deckADT would have its own create method. Without modules, we'd have to rely on naming conventions to keep the operations straight. For example, all of the operations on cards might begin with card_ while those for decks would start with deck_. Thus, we would have separate functions, card_create and deck_create.

2.3 ADTs and Objects

As we have seen, an ADT comprises a set of operations that manipulate some underlying data representation. This should sound familiar to you. If you are working in an object-oriented language (such as Python), then it is natural to think of implementing an ADT as an object, since objects also combine data and operations. Simply put, an object "knows stuff (data) and does stuff (operations)." The data in an object is stored in instance variables, and the operations are its methods. We can use the instance variables to store the concrete representation of an ADT and write methods to implement the operations.

As you know, new object types are defined using the `class` mechanism. As the Python language has evolved, it has come to support two different kinds of classes sometimes called the *classic* and *new-style* classes. For our examples, classic and new-style classes behave exactly the same. We will use Python's new-style classes throughout this book as they are strongly recommended for new code. A new-style class is indicated simply by having the class inherit from the built-in class `object`. You do not need to know any details about inheritance in order to use new-style classes; you just need to change the class heading slightly. For example, to create a `Card` class with new-style classes, we write `class Card(object):` instead of `class Card:`.[1]

2.3.1 Specification

In object-oriented languages, new object data types can be created by defining a new class. We can turn an ADT description directly into an appropriate class specification. Here is a class specification for our `Card` example:

```
class Card(object):
    """A simple playing card. A Card is characterized by two components.
     rank: an integer value in the range 1-13, inclusive (Ace-King)
     suit: a character in 'cdhs' for clubs, diamonds, hearts, and
           spades."""

    def __init__(self, rank, suit):
        """Constructor
        pre: rank in range(1,14) and suit in 'cdhs'
        post: self has the given rank and suit"""
```

[1]In Python 3.0, support for classic classes has been dropped and either class heading form will produce a new-style class.

```
    def suit(self):
        """Card suit
        post: Returns the suit of self as a single character"""

    def rank(self):
        """Card rank
        post: Returns the rank of self as an int"""

    def suitName(self):
        """Card suit name
        post: Returns one of ('Clubs', 'Diamonds', 'Hearts',
              'Spades') corrresponding to self's suit."""

    def rankName(self):
        """Card rank name
        post: Returns one of ('Ace', 'Two', 'Three', ..., 'King')
              corresponding to self's rank."""

    def __str__(self):
        """String representation
        post: Returns string representing self, e.g. 'Ace of Spades' """
```

Basically, this specification is just the outline of a `Card` class as it would look in Python. The docstring for the class gives an overview, and the docstrings for the methods specify what each one does. Following Python conventions, the method names that begin and end with double underscores (`__init__` and `__str__`) are special. Python recognizes `__init__` as the constructor, and the `__str__` method will be called whenever Python is asked to convert a `Card` object into a string. For example:

```
>>> c = Card(4,'c')
>>> print c
Four of Clubs
```

We have now translated our ADT into an object-oriented form. Clients of this class will use dot notation to perform operations on the ADT. Here's the code that prints out all 52 cards translated into its object-based form:

```
# printcards.py
#    Simple test of the Card ADT

from Card import Card

def printAll():
    for suit in 'cdhs':
        for rank in range(1,14):
            card = Card(rank, suit)
            print 'Rank:', card.rank()
            print 'Suit:', card.suit()
            print card

if __name__ == '__main__':
    printAll()
```

Notice that the constructor is invoked by using the name of the class, Card, and the `__str__` method is implicitly called by Python when it is asked to print the card.

2.3.2 Implementation

We can translate our previous implementation of the card ADT into our new class-based implementation. Now the rank and suit components of a card can just be stored in appropriate instance variables:

```
# Card.py
class Card(object):
    """A simple playing card. A Card is characterized by two components:
    rank: an integer value in the range 1-13, inclusive (Ace-King)
    suit: a character in 'cdhs' for clubs, diamonds, hearts, and
    spades."""

    SUITS = 'cdhs'
    SUIT_NAMES = ['Clubs', 'Diamonds', 'Hearts', 'Spades']

    RANKS = range(1,14)
    RANK_NAMES = ['Ace', 'Two', 'Three', 'Four', 'Five', 'Six',
                  'Seven', 'Eight', 'Nine', 'Ten',
                  'Jack', 'Queen', 'King']

    def __init__(self, rank, suit):
        """Constructor
        pre: rank in range(1,14) and suit in 'cdhs'
        post: self has the given rank and suit"""

        self.rank_num = rank
        self.suit_char = suit
    def suit(self):
        """Card suit
        post: Returns the suit of self as a single character"""

        return self.suit_char

    def rank(self):
        """Card rank
        post: Returns the rank of self as an int"""

        return self.rank_num

    def suitName(self):
        """Card suit name
        post: Returns one of ('clubs', 'diamonds', 'hearts',
              'spades') corresponding to self's suit."""

        index = self.SUITS.index(self.suit_char)
        return self.SUIT_NAMES[index]

    def rankName(self):
        """Card rank name
        post: Returns one of ('ace', 'two', 'three', ..., 'king')
              corresponding to self's rank."""

        index = self.RANKS.index(self.rank_num)
        return self.RANK_NAMES[index]

    def __str__(self):
        """String representation
        post: Returns string representing self, e.g. 'Ace of Spades' """

        return self.rankName() + ' of ' + self.suitName()
```

Notice that the lookup tables from the previous version have now been implemented as variables that are assigned inside of the `Card` class but outside of any of the methods of the class. These are *class variables*. They "live" inside the class definition, so there is one copy shared by all instances of the class. These variables are accessed just like instance variables using the `self.<name>` convention. When Python is asked to retrieve the value of an object's attribute, it first checks to see if the attribute has been assigned directly for the object. If not, it will look in the object's class to find it. For example, when the `suitName` method accesses `self.SUITS`, Python sees that `self` does not have a `SUIT` attribute, so the value from the `Card` class is used (because `self` is a `Card`).

You now have three different kinds of variables for storing information in programs: regular (local) variables, instance variables, and class variables. Choosing the right kind of variable for a given piece of information is an important decision when implementing ADTs. The first question you must answer is whether the data needs to be remembered from one method invocation to another. If not, you should use a local variable. The `index` variable used in `rankName()` is a good example of a local variable; its value is no longer needed once the method terminates. Notice that there is also a local variable called `index` in the `suitName` method. These are two completely independent variables, even though they happen to have the same name. Each exists only while the method where they are used is executing. We could have written this code using an instance variable `self.index` in these two methods. Doing so would be a misleading design choice, because we have no reason to hang onto the value of index from the last execution of `rankName` or `suitName`. Reusing an instance variable in this case would imply a connection where none exists.

Data that does need to be remembered from one method invocation to another should be stored in either instance variables or class variables. The decision about which to use in this case depends on whether the data may be different from one object to the next or whether it is the same for all objects of the class. In our card example, `self.rank_num` and `self.suit_char` are values that will vary among cards. They are part of the intrinsic state of a particular card, so they have to be instance variables. The suit names, on the other hand will be the same for all cards of the class, so it makes sense to use a class variable for that. Constants are often good candidates for class variables, since, by definition, they are the same from one object to the next. However, there are also times when non-constant class variables make sense. Keeping these simple rules in mind should help you turn your ADTs into working classes.

As you can see there is a natural correspondence between the notion of an ADT and an object-oriented class. When using an object-oriented language, you will usually want to implement an ADT as a class. The nice thing about using classes is that they naturally combine the two facets of an ADT (data and operations) into a single programming structure.

2.3.3 Changing the Representation

We have emphasized that the primary strength of using ADTs to design software is implementation independence. However, the playing card example that we've discussed so far has not really illustrated this point. After all, we said that a card has a rank that is an `int` and a suit that is a character, then we simply stored these values as instance variables. Isn't the client directly using the representation when it manipulates suits and ranks?

The reason it *seems* that the client has access to the representation in this case is simply because the concrete representation that we've chosen directly mirrors the data types that are used to pass information to and from the ADT. However, since access to the data takes place through methods (like `suit` and `rank`) we can actually change the concrete representation without affecting the client code. This is where the independence comes in.

Suppose we are developing card games for a handheld device such as a PDA or cell phone. On such a device, we might have strict memory limitations. Our current representation of cards requires two instance variables for each card; the rank, which is a 32-bit `int`; and the suit, which is a character. An alternative way to think about cards is simply to number them. Since there are 52 cards, each can be represented as a number from 0 to 51. Think of putting the cards in order so that all the clubs come first, diamonds second, etc. Within each suit, put the cards in rank order. Now we have a complete ordering where the first card in the deck is the ace of clubs, and the last card is the king of spades.

Given a card's number, we can calculate its rank and suit. Since there are 13 cards in each suit, dividing the card number by 13 (using integer division) produces a value between 0 and 3 (inclusive). Clubs will yield a 0, diamonds a 1, etc. Furthermore, the remainder from the division will give the relative position of the card within the suit (i.e., its rank). For example, if the card number is 37, $37//13 = 2$ so the suit is hearts, and $37\%13 = 11$ which corresponds to a rank of queen since the first card in a suit (the ace) will have a remainder of 0. So card 37 is the queen of hearts. Using this approach, the concrete representation of our `Card` ADT can be a single number. We leave it as an exercise for the reader to complete an implementation of the `Card` class using this more-memory-efficient alternative representation.

2.3.4 Object-Oriented Design and Programming

As you have seen, there is a close correspondence between the ideas of ADTs and object-oriented programming. But there is more to object-orientation (OO) than just implementing ADTs. Most OO gurus talk about three features that together make development truly object-oriented: encapsulation, polymorphism, and inheritance.

Encapsulation

As you know, objects know stuff and do stuff. They combine data and operations. This process of packaging some data along with the set of operations that can be performed on the data is called *encapsulation*.

Encapsulation is one of the major attractions of using objects. It provides a convenient way to compose solutions to complex problems that corresponds to our intuitive view of how the world works. We naturally think of the world around us as consisting of interacting objects. Each object has its own identity, and knowing what kind of object it is allows us to understand its nature and capabilities. When you look out your window, you see houses, cars, and trees, not a swarming mass of countless molecules or atoms.

From a design standpoint, encapsulation also provides the critical service of separating the concerns of "what" vs. "how." The actual implementation of an object is independent of its use. Encapsulation is what gives us implementation independence. Encapsulation is probably the chief benefit of using objects, but alone it only makes a system *object-based.* To be truly objected-*oriented*, the approach must also have the characteristics of polymorphism and inheritance.

Polymorphism

Literally, the word *polymorphism* means "many forms." When used in object-oriented literature, this refers to the fact that what an object does in response to a message (a method call) depends on the type or class of the object. Consider a simple example. Suppose you are working with a graphics library for drawing two-dimensional shapes. The library provides a number of primitive geometric shapes that can be drawn into a window on the screen. Each shape has an operation that actually draws the shape. We have a collection of classes something like this:

```
class Circle(object):
    def draw(self, window):
        # code to draw the circle

class Rectangle(object):
    def draw(self, window):
        # code to draw the rectangle

class Polygon(object):
    def draw(self, window):
        # code to draw the polygon
```

Of course, each of these classes would have other methods in addition to its draw method. Here we're just giving a basic outline for illustration.

Suppose you write a program that creates a list containing a mixture of geometric objects: circles, rectangles, polygons, etc. To draw all of the objects in the list, you would write code something like this:

```
for obj in objects:
    obj.draw(win)
```

Now consider the single line of code in the loop body. What function is called when obj.draw(win) executes? Actually, this single line of code calls several distinct functions. When obj is a circle, it executes the draw method from the circle class. When obj is a rectangle, it is the draw method from the rectangle class, and so on. The draw operation takes many forms; the particular one used depends on the type of obj. That's the polymorphism.

Polymorphism gives object-oriented systems the flexibility for each object to perform an action just the way that it should be performed for that object. If we didn't have objects that supported polymorphism we'd have to do something like this:

```
for obj in objects:
   if type(obj) is Circle:
       draw_circle(...)
   elif type(obj) is Rectangle:
       draw_rectangle(...)
   elif type(obj) is Polygon:
       draw_polygon(...)
   ...
```

Not only is this code more cumbersome, it is also much less flexible. If we want to add another type of object to our library, we have to find all of the places where we made a decision based on the object type and add another branch. In the polymorphic version, we can just create another class of geometric object that has its own draw method, and all the rest of the code remains exactly the same. Polymorphism allows us to extend the program without having to go in and modify the existing code.

Inheritance

The third important property for object-oriented development is *inheritance*. As its name implies, the idea behind inheritance is that a new class can be defined to borrow behavior from another class. The new class (the one doing the borrowing) is called a *subclass*, and the existing class (the one being borrowed from) is its *superclass*.

For example, if we are building a system to keep track of employees, we might have a class `Employee` that contains the general information and methods that are common to all employees. One sample attribute would be a `homeAddress` method that returns the home address of an employee. Within the class of all employees, we might distinguish between `SalariedEmployee` and `HourlyEmployee`. We could make these subclasses of `Employee`, so they would share methods like `homeAddress`; however, each subclass would have its own `monthlyPay` function, since pay is computed differently for these different classes of employees. Figure 2.1 shows a simple class diagram depicting this situation. The arrows with open heads indicate inheritance; the subclasses inherit the `homeAddress` method defined in the `Employee` class, but each defines its own implementation of the `monthlyPay` method.

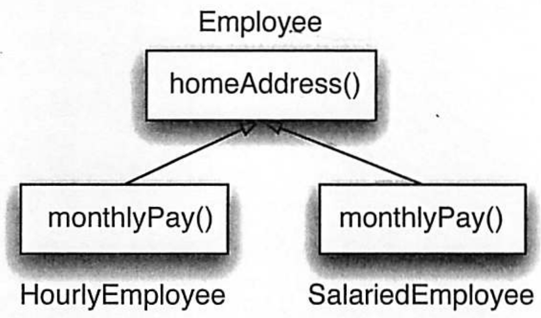

Figure 2.1: Simple example of inheritance with subclasses inheriting one shared method and each separately implementing one method

Inheritance provides two benefits. One is that we can structure the classes of a system to avoid duplication of operations. We don't have to write a separate `homeAddress` method for the `HourlyEmployee` and `SalariedEmployee` classes. A closely related benefit is that new classes can often be based on existing classes, thus promoting code reuse.