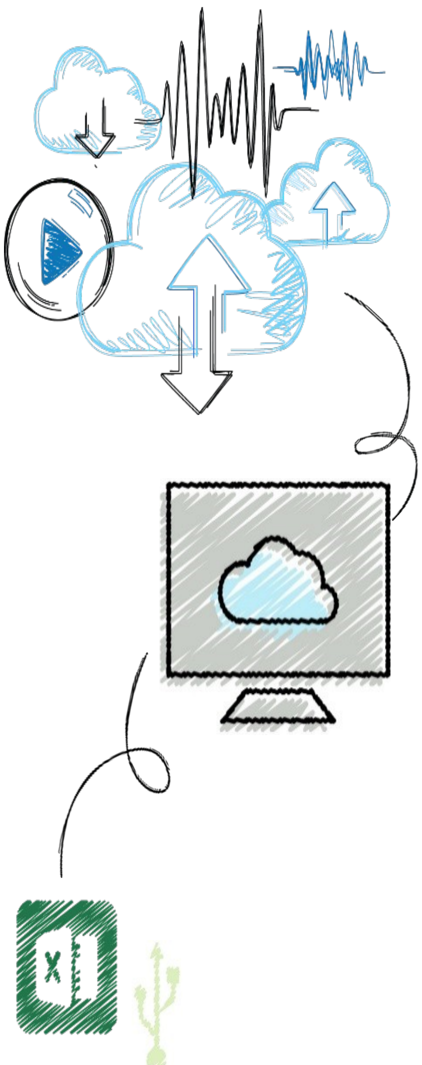
Fundamentals of Python: First Programs Second Edition

Chapter 9

Design with Classes



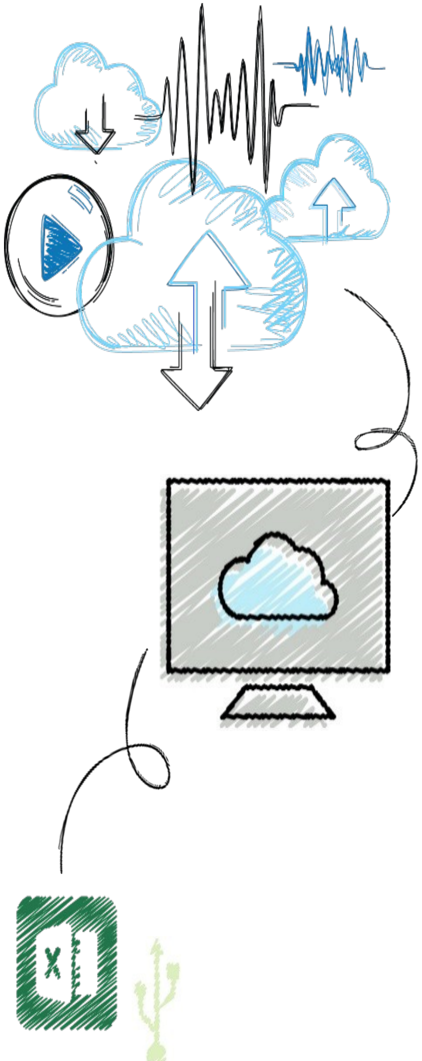
Objectives (1 of 2)

9.1 Determine the attributes and behavior of a class of objects required by a program

9.2 List the methods, including their parameters and return types, that realize the behavior of a class of objects

9.3 Choose the appropriate data structures to represent the attributes of a class of objects

9.4 Define a constructor, instance variables, and methods for a class of objects



Objectives (2 of 2)

9.5 Recognize the need for a class variable

9.6 Define a method that returns the string representation of an object

9.7 Define methods for object equality and comparisons

9.8 Exploit inheritance and polymorphism when developing classes

9.9 Transfer objects to and from files

Getting Inside Objects and Classes

- Programmers who use objects and classes know:
  - The interface that can be used with a class
  - The state of an object
  - How to instantiate a class to obtain an object
- Objects are abstractions
  - Package their state and methods in a single entity that can be referenced with a name
- Class definition is like a blueprint for each of the objects of that class and contains:
  - Definitions of all of the methods that its objects recognize
  - Descriptions of the data structures used to maintain the state of an object

A First Example: The Student Class (1 of 3)

- A course-management application needs to represent information about students in a course

```
>>> from student import Student
>>> s = Student("Maria", 5)
>>> print(s)
Name: Maria
Scores: 0 0 0 0 0
>>> s.setScore(1, 100)
>>> print(s)
Name: Maria
Scores: 100 0 0 0 0
>>> s.getHighScore()
100
>>> s.getAverage()
20
>>> s.getScore(1)
100
>>> s.getName()
'Maria'
```

A First Example: The Student Class (2 of 3)

Student Method	What It Does
s = Student(name, number)	Returns a Student object with the given name and number of scores. Each score is initially 0
s.getName()	Returns the student's name
s.getScore(i)	Returns the student's *i*th score, i must range from 1 through the number of scores
s.setScore(i, score)	Resets the student's *i*th score to score, i must range from 1 through the number of scores
s.getAverage()	Returns the student's average score
s.getHighScore()	Returns the student's highest score
s.__str()__	Same as str(s). Returns a string representation of the student's information

A First Example: The Student Class (3 of 3)

- Syntax of a simple class definition:

  class <class name>(<parent class name>):
  <method definition−1>
  ...
  <method definition−n>

  - Class name is a Python identifier
    - Typically capitalized
- Python classes are organized in a tree-like **class hierarchy**
  - At the top, or root, of this tree is the **object** class
  - Some terminology: **subclass**, **parent class**

Docstrings

- Docstrings can appear at three levels:
  - Module
  - Just after class header
    - To describe its purpose
  - After each method header
    - Serve same role as they do for function definitions
- **help(Student)** prints the documentation for the class and all of its methods

Method Definitions

- Method definitions are indented below class header
- Syntax of method definitions similar to functions
  - Can have required and/or default arguments, return values, create/use temporary variables
  - Returns **None** when no **return** statement is used
- Each method definition must include a first parameter named **self**
- Example: **s.getScore(4)**
  - Binds the parameter **self** in the method **getScore** to the **Student** object referenced by the variable **s**

The _init_ Method and Instance Variables

- Most classes include the **__init__** method

```
def __init__(self, name, number):
    """All scores are initially 0."""
    self.name = name
    self.scores = []
    for count in range(number):
        self.scores.append(0)
```

  - Class's **constructor**
  - Runs automatically when user instantiates the class
- Example: **s = Student("Juan", 5)**
- **Instance variables** represent object attributes
  - Serve as storage for object state
  - Scope is the entire class definition

CENGAGE

The _str_ Method

- Classes usually include an **__str__** method
  - Builds and returns a string representation of an object’s state

```
def __str__(self) :
    “““Returns the string representation of the student.”””
    return "Name: " + self.name + “\nScores: ” + \
           “ ”.join(map(str, self.scores))
```

- When **str** function is called with an object, that object’s **__str__** method is automatically invoked
- Perhaps the most important use of **__str__** is in debugging

Accessors and Mutators

- Methods that allow a user to observe but not change the state of an object are called **accessors**
- Methods that allow a user to modify an object’s state are called **mutators**

```
def setScore(self, i, score):
"""“Resets the ith score, counting from 1.”""
self.scores[i − 1] = scor
```

- Tip: if there’s no need to modify an attribute (e.g., a student’s name), do not include a method to do that

CENGAGE

The Lifetime of Objects (1 of 2)

- The lifetime of an object's instance variables is the lifetime of that object
- An object becomes a candidate for the graveyard when it can no longer be referenced

```
>>> s = Student("Sam", 10)
>>> cscilll = [s]
>>> cscilll
[<__main__.Student instance at 0xllba2b0>]
>>> s
<__main__.Student instance at 0xllba2b0>

>>> s = None
>>> cscilll.pop()
<__main__.Student instance at 0xllba2b0>
>>> print(s)
None
>>> cscilll
[]
```

The Lifetime of Objects (2 of 2)

- From previous code, the student object still exists
  - But the Python virtual machine will eventually recycle its storage during a process called **garbage collection**

Rules of Thumb for Defining a Simple Class

- Before writing a line of code, think about the behavior and attributes of the objects of new class
- Choose an appropriate class name and develop a short list of the methods available to users
- Write a short script that appears to use the new class in an appropriate way
- Choose appropriate data structures for attributes
- Fill in class template with **__init__** and **__str__**
- Complete and test remaining methods incrementally
- Document your code

Data-Modeling Examples

- As you have seen, objects and classes are useful for modeling objects in the real world
- In this section, we explore several other examples

Rational Numbers

- **Rational number** consists of two integer parts, a numerator and a denominator
  - Examples: 1/2, 2/3, etc.
- Python has no built-in type for rational numbers
  - We will build a new class named **Rational**

```
>>> oneHalf = Rational(1, 2)
>>> oneSixth = Rational(1, 6)
>>> print(oneHalf)
1/2
>>> print(oneHalf + oneSixth)
2/3
>>> oneHalf == oneSixth
False
>>> oneHalf > oneSixth
True
```

Rational Number Arithmetic and Operator Overloading (1 of 3)

Operator	Method Name
+	_add_
-	_sub_
*	_mul_
/	_div_
%	_mod_

- Object on which the method is called corresponds to the left operand
  - For example, the code **x + y** is actually shorthand for the code **x.__add__(y)**

Rational Number Arithmetic and Operator Overloading (2 of 3)

- To overload an arithmetic operator, you define a new method using the appropriate method name
- Code for each method applies a rule of rational number arithmetic

Type of Operation	Rule
Addition	$\frac{n_1}{d_1} + \frac{n_2}{d_2} = \frac{(n_1 d_2 + n_2 d_1)}{d_1 d_2}$
Subtraction	$\frac{n_1}{d_1} - \frac{n_2}{d_2} = \frac{(n_1 d_2 - n_2 d_1)}{d_1 d_2}$
Multiplication	$\frac{a}{b} \cdot \frac{c}{d} = \frac{ac}{bd}$
Division	$\frac{\frac{n_1}{d_1}}{\frac{n_2}{d_2}} = \frac{(n_1 d_2)}{d_1 n_2}$

CENGAGE

Rational Number Arithmetic and Operator Overloading (3 of 3)

```
def __add__(self, other):
    """Returns the sum of the numbers.
    self is the left operand and other is
    the right operand."""
    newNumer = self.numer * other.denom + \
               other.numer * self.denom
    newDenom = self.denom * other.denom
    return Rational(newNumer, newDenom)
```

- **Operator overloading** is another example of an abstraction mechanism
  - We can use operators with single, standard meanings even though the underlying operations vary from data type to data type

Comparison Methods

Operator	Meaning	Method
==	Equals	_eq_
!=	Not equals	_ne_
<	Less than	_lt_
<=	Less than or equal	_le_
>	Greater than	_gt_
>=	Greater than or equal	_ge_

Equality and the _eq_ Method

- Not all objects are comparable using < or >, but any two objects can be compared for == or !=

twoThirds < "hi there" should generate an error

twoThirds != "hi there" should return **True**

```
def __eq__(self, other):
    """Tests self and other for equality."""
    if self is other: # Object identity?
        return True
    elif type(self) != type(other): # Types match?
        return False
    else:
        return self.numer == other.numer and \
            self.denom == other.denom
```

- Include **__eq__** in any class where a comparison for equality uses a criterion other than object identity

Savings Accounts and Class Variables (1 of 4)

SavingsAccount Method	What It Does
a = SavingsAccount(name, pin, balance = 0.0)	Returns a new account with the given name, PIN, and balance
a.deposit(amount)	Deposits the given amount to the account's balance
a.withdraw(amount)	Withdraws the given amount from the account's balance
a.getBalance()	Returns the account's balance
a.getName()	Returns the account's name
a.getPin()	Returns the account's PIN
a.computeInterest()	Computes the account's interest and deposits it
a.__str__()	Same as str(a). Returns the string representation of the account

Savings Accounts and Class Variables (2 of 4)

- Code for SavingsAccount:

```
"""
File: savingsaccount.py
This module defines the SavingsAccount class.
"""
class SavingsAccount(object):
 """This class represents a savings account
   with the owner's name, PIN, and balance."""

RATE = 0.02 # Single rate for all accounts
def __init__(self, name, pin, balance = 0.0):
   self.name = name
   self.pin = pin
   self.balance = balance
def __str__(self) :
 """Returns the string rep."""
   result = 'Name: ' + self.name + '\n'
   result += 'PIN: ' + self.pin + '\n'
   result += 'Balance: ' + str(self.balance)
   return result
```

Savings Accounts and Class Variables (3 of 4)

- Code for SavingsAccount (continued):

```
def getBalance(self):
    “““Returns the current balance.”””
    return self.balance

def getName(self):
    “““Returns the current name.”””
    return self.name

def getPin(self):
   “““Returns the current pin.”””
    return self.pin

def deposit(self, amount):
    “““Deposits the given amount and returns None.”””
    self.balance += amount
    return None
```

CENGAGE

Savings Accounts and Class Variables (4 of 4)

- Code for SavingsAccount (continued):

```
def withdraw(self, amount):
    “““Withdraws the given amount.
    Returns None if successful, or an
    error message if unsuccessful. ”””
    if amount < 0:
        return “Amount must be >= 0”
    elif self.balance < amount:
        return “Insufficient funds”
    else:
        self.balance -= amount
        return None

def computeInterest(self):
    “““Computes, deposits, and returns the interest.”””
    interest = self.balance * SavingsAccount.RATE
    self.deposit(interest)
    return interest
```

Putting the Accounts into a Bank (1 of 3)

```
>>> from bank import Bank
>>> from savingsaccount import SavingsAccount
>>> bank = Bank()
>>> bank.add(SavingsAccount(“Wilma”, “1001”, 4000.00))
>>> bank.add(SavingsAccount(“Fred”, “1002”, 1000.00))
>>> print(bank)
Name: Fred
PIN: 1002
Balance: 1000.00
Name: Wilma
PIN: 1001
Balance: 4000.00
>>> account = bank.get(“Wilma”, “1000”)
>>> print(account)
None
```

Putting the Accounts into a Bank (2 of 3)

```
>>> account = bank.get(“Wilma”, “1001”)
>>> print (account)
Name: Wilma
PIN: 1001
Balance: 4000.00
>>> account.deposit(25.00)
>>> print(account)
Name: Wilma
PIN: 1001
Balance: 4025.00
>>> print(bank)
Name: Fred
PIN: 1002
Balance: 1000.00
Name: Wilma
PIN: 1001
Balance: 4025.00
```

Putting the Accounts into a Bank (3 of 3)

Bank Method	What It Does
b = Bank()	Returns a bank
b.add(account)	Adds the given account to the bank
b.remove(name, pin)	Removes the account with the given name and pin from the bank and returns the account. If the account is not in the bank, returns None
b.get(name, pin)	Returns the account associated with the name and pin if it's in the bank. Otherwise, returns None
b.computeInterest()	Computes the interest on each account, deposits it in that account, and returns the total interest
b.__str__()	Same as str(b). Returns a string representation of the bank (all the accounts)

Using pickle for Permanent Storage of Objects

- **pickle** allows programmer to save and load objects using a process called **pickling**
  - Python takes care of all of the conversion details

```
import pickle

def save(self, fileName = None):
   """Saves pickled accounts to a file. The parameter
   allows the user to change filenames."""
   if fileName != None:
      self.fileName = fileName
   elif self.fileName == None:
      return
   fileObj = open(self. fileName, "wb")
   for account in self.accounts.values():
      pickle.dump(account, fileObj)
   fileObj.close()
```

Input of Objects and the try-except Statement

```
try:
    <statements>
except <exception type>:
    <statements>

def __init__(self, fileName = None):
    """Creates a new dictionary to hold the accounts.
    If a filename is provided, loads the accounts from
    a file of pickled accounts."""
    self.accounts = {}
    self.fileName = fileName
    if fileName != None:
        fileObj = open(fileName, "rb")
        while True:
            try:
                account = pickle.load(fileObj)
                self.add(account)
            except EOFError:
                fileObj.close()
                break
```

CENGAGE

Playing Cards (1 of 4)

- Use of the **Card** class:

```
>>> threeOfSpades = Card(3, "Spades")
>>> jackOfSpades = Card(11, "Spades")
>>> print(jackOfSpades)
Jack of Spades
>>> threeOfSpades.rank < jackOfSpades.rank
True
>>> print(jackOfSpades.rank, jackOfSpades.suit)
11 Spades
```

- Because the attributes are only accessed and never modified, we do not include any methods other than **__str__** for string representation
- A card is little more than a container of two data values

Playing Cards (2 of 4)

```
class Card(object):
   """A card object with a suit and rank."""

   RANKS = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13)
   SUITS = ("Spades", "Diamonds", "Hearts", "Clubs")

 def __init__(self, rank, suit):
 """Creates a card with the given rank and suit. """
       self.rank = rank
       self.suit = suit

 def __str__(self) :
 """Returns the string representation of a card. """
      if self.rank == 1:
         rank = "Ace"
      elif self.rank == 11:
         rank = "Jack"
      elif self.rank == 12:
         rank = "Queen"
      elif self.rank == 13:
         rank = "King"
      else:
         rank = self.rank
      return str(rank) + "of " + self.suit
```

Playing Cards (3 of 4)

- Unlike an individual card, a deck has significant behavior that can be specified in an interface
- One can shuffle the deck, deal a card, and determine the number of cards left in it

```
>>> deck = Deck()
>>> print(deck)
--- the print reps of 52 cards, in order of suit and rank
>>> deck.shuffle()
>>> len(deck)
52
>>> while len(deck) > 0:
        card = deck.deal()
        print(card)
--- the print reps of 52 randomly ordered cards
>>> len(deck)
0
```

Playing Cards (4 of 4)

Deck Method	What It Does
d = Deck()	Returns a deck
d.__len__()	Same as len(d). Returns the number of cards currently in the deck
d.shuffle()	Shuffles the cards in the deck
d.deal()	If the deck is not empty, removes and returns the topmost card. Otherwise, returns None
d.__str__()	Same as str(d). Returns a string representation of the deck (all the cards in it)

Building a New Data Structure: The Two-Dimensional Grid

- A useful data structure: **two-dimensional grid**
- A grid organizes items by position in rows and columns
- In this section, we develop a new class called **Grid**
  - For applications that require grids

The Interface of the Grid Class (1 of 2)

- The constructor or operation to create a grid allows you to specify the width, the height, and an optional initial fill value for all of the positions
  - Default fill value is None
- You access or replace an item at a given position by specifying the row and column of that position, using the notation:
  - **grid[<row>] [<column>]**

The Interface of the Grid Class (2 of 2)

Grid Method	What It Does
g = Grid(rows, columns, fillValue = None)	Returns a new Grid object
g.getHeight()	Returns the number of rows
g.getWidth()	Returns the number of columns
g.__str__()	Same as str(g). Returns the string representation
g.__getitem__(row)[column]	Same as g[row][column]
g.find(value)	Returns (row, column) if value is found, or None otherwise

The Implementation of the Grid Class: Instance Variables for the Data

- Implementation of a class provides the code for the methods in its interface
  - As well as the instance variables needed to track the data contained in objects of that class
- Next step is to choose the data structures that will represent the two-dimensional structure within a **Grid** object
  - A single instance variable named **self.data** holds the top-level list of rows
  - Each item within this list will be a list of the columns in that row
- Other two methods:
  - _init_, which initializes the instance variables
  - _str_, which allows you to view the data during testing

The Implementation of the Grid Class: Subscript and Search

- Subscript operator
  - used to access an item at a grid position or to replace it there
- In the case of access
  - The subscript appears within an expression, as in **grid[1] [2]**
- Search operation named **find** must loop through the grid's list of lists
  - Until it finds the target item or runs out of items to examine

Structuring Classes with Inheritance and Polymorphism

- Most object-oriented languages require the programmer to master the following techniques:
  - **Data encapsulation:** Restricting manipulation of an object's state by external users to a set of method calls
  - **Inheritance:** Allowing a class to automatically reuse/ and extend code of similar but more general classes
  - **Polymorphism:** Allowing several different classes to use the same general method names
- Python's syntax doesn't enforce data encapsulation
- Inheritance and polymorphism are built into Python

Inheritance Hierarchies and Modeling (1 of 2)

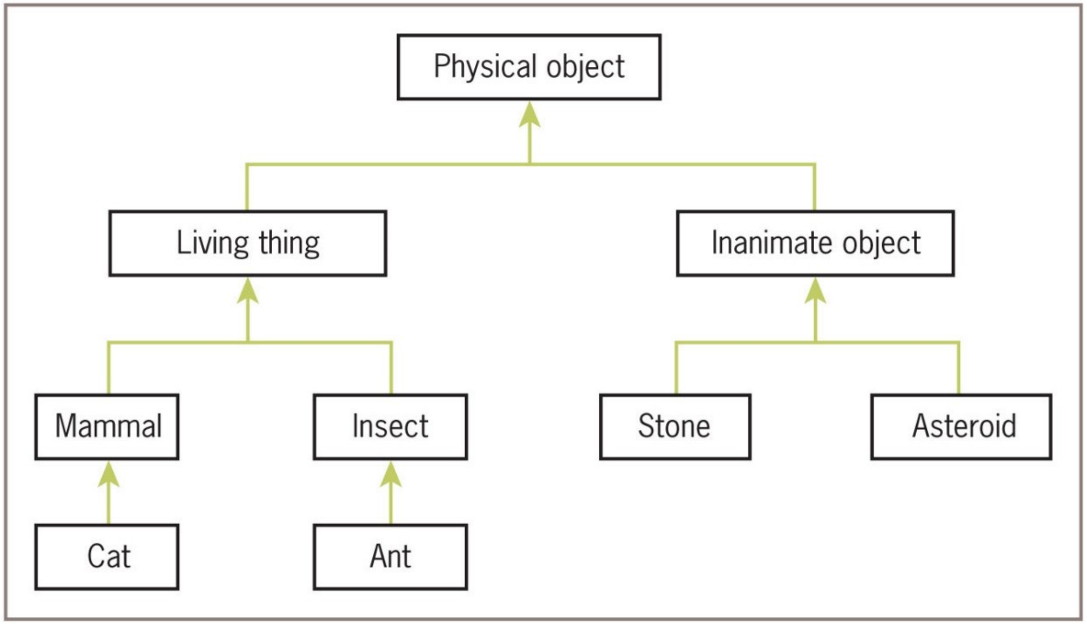


Figure 9-5 A simplified hierarchy of objects in the natural world

Inheritance Hierarchies and Modeling (2 of 2)

- In Python, all classes automatically extend the built-in **object** class
- It is possible to extend any existing class:

  class <new class name>(<existing parent class name>):

- Example:
  - **PhysicalObject** would extend **object**
  - **LivingThing** would extend **PhysicalObject**
- Inheritance hierarchies provide an abstraction mechanism that allows the programmer to avoid reinventing the wheel or writing redundant code

Example 1: A Restricted Savings Account

```
>>> account = RestrictedSavingsAccount(“Ken”, “1001”, 500.00)
>>> print(account)
Name: Ken
PIN: 1001
Balance: 500.0
>>> account.getBalance()
500.0
>>> for count in range(3):
account.withdraw(100)
>>> account.withdraw(50)
‘No more withdrawals this month’
>>> account.resetCounter()
>>> account.withdraw(50)
```

- To call a method in the parent class from within a method with the same name in a subclass:

```
<parent class name>.<method name>(self, <other arguments>)
```

CENGAGE

Example 2: The Dealer and a Player in the Game of Blackjack (1 of 2)

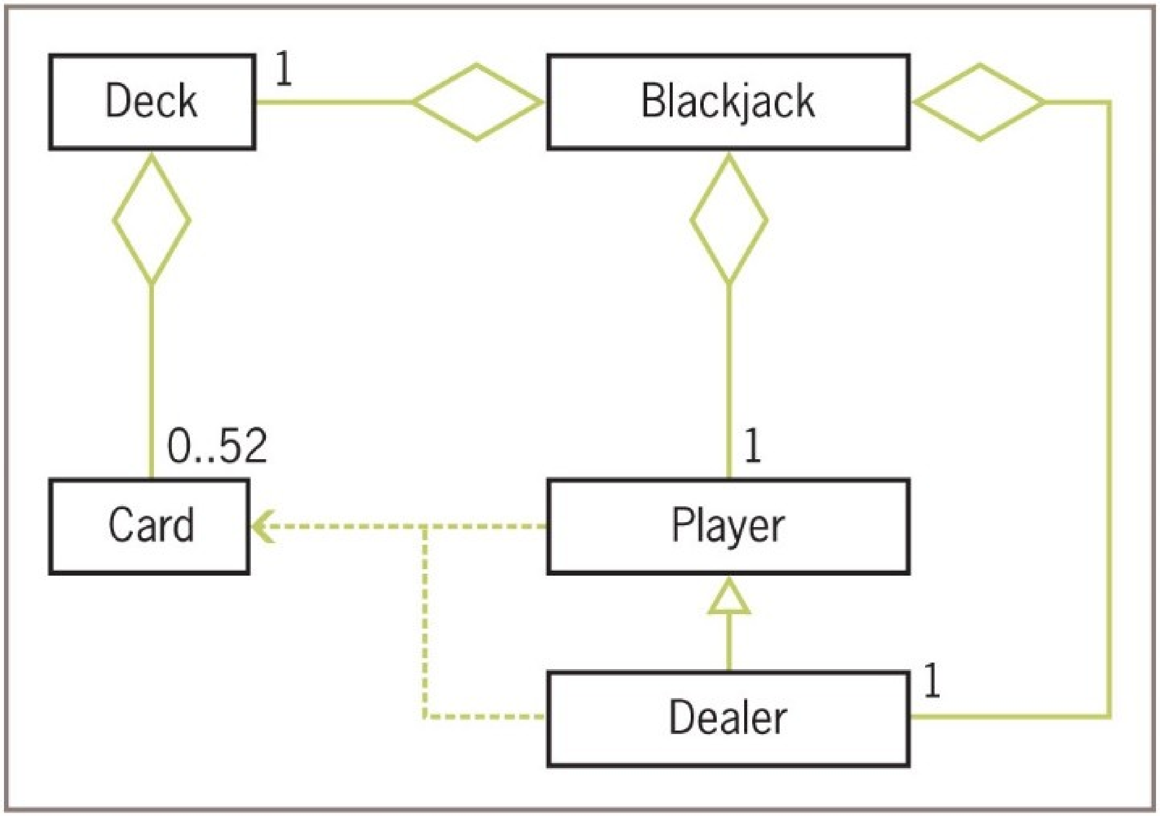


Figure 9-6 The classes in the blackjack game application

Example 2: The Dealer and a Player in the Game of Blackjack (2 of 2)

- An object belonging to **Blackjack** class sets up the game and manages the interactions with user

```
>>> from blackjack import Blackjack
>>> game = Blackjack()
>>> game.play()
Player:
2 of Spades, 5 of Spades
7 points Dealer:
5 of Hearts
Do you want a hit? [y/n]: y
Player:
2 of Spades, 5 of Spades, King of Hearts
17 points
Do you want a hit? [y/n]: n
Dealer:
5 of Hearts, Queen of Hearts, 7 of Diamonds
22 points
Dealer busts and you win
```

Polymorphic Methods

- We subclass when two classes share a substantial amount of **abstract behavior**
  - The classes have similar sets of methods/operations
  - A subclass usually adds something extra
- The two classes may have the same interface
  - One or more methods in subclass override the definitions of the same methods in the superclass to provide specialized versions of the abstract behavior
    - **Polymorphic methods** (e.g., the **__str__** method)

The Costs and Benefits of Object-Oriented Programming (1 of 2)

- **Imperative programming**
  - Code consists of I/O, assignment, and control (selection/iteration) statements
  - Does not scale well
- Improvement: Embedding sequences of imperative code in function definitions or subprograms
  - **Procedural programming**
- **Functional programming** views a program as a set of cooperating functions
  - No assignment statements

The Costs and Benefits of Object-Oriented Programming (2 of 2)

- Functional programming does not conveniently model situations where data must change state
- Object-oriented programming attempts to control the complexity of a program while still modeling data that change their state
  - Divides up data into units called objects
  - Well-designed objects decrease likelihood that system will break when changes are made within a component
  - Can be overused and abused

Chapter Summary (1 of 2)

- A simple class definition consists of a header and a set of method definitions
- In addition to methods, a class can also include instance variables
- Constructor or **__init__** method is called when a class is instantiated
- A method contains a header and a body
- An instance variable is introduced and referenced like any other variable, but is always prefixed with **self**
- Some standard operators can be overloaded for use with new classes of objects
- When a program can no longer reference an object, it is considered dead and its storage is recycled by the garbage collector

Chapter Summary (2 of 2)

- A class variable is a name for a value that all instances of a class share in common
- Pickling is the process of converting an object to a form that can be saved to permanent file storage
- **try-except** statement is used to catch and handle exceptions
- Most important features of OO programming: encapsulation, inheritance, and polymorphism
  - Encapsulation restricts access to an object's data to users of the methods of its class
  - Inheritance allows one class to pick up the attributes and behavior of another class for free
  - Polymorphism allows methods in several different classes to have the same headers
- A data model is a set of classes that are responsible for managing the data of a program