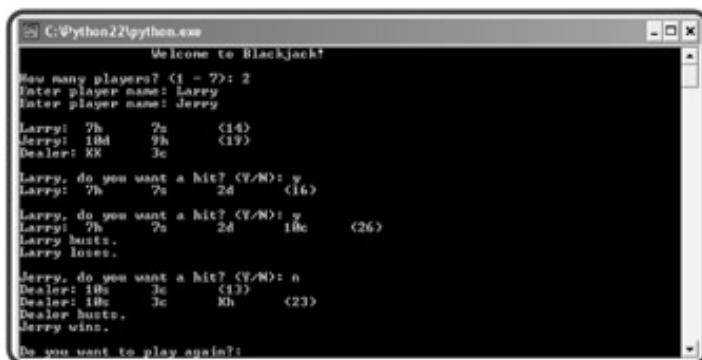


Introducing the Blackjack Game

The final project for this chapter is a simplified version of the card game, black-jack. The game works like this: Players are dealt cards with point values. Each player tries to reach a total of 21 without going over. Numbered cards count as their face value. An ace counts as either 1 or 11 (whichever is best for the player) and any jack, queen, or king counts as 10.

The computer is the dealer and competes against one to seven players. At the opening of the round, the computer deals all participants (including itself) two cards. Players can see all of their cards, and the computer even displays their total. However, one of the dealer's cards is hidden for the time being.

Next, each player gets a chance to take additional cards. Each player can take one card at a time for as long as the player likes. But if the player's total goes over 21 (known as "busting"), the player loses. If all players bust, the computer reveals its first card and the round is over. Otherwise, play continues. The computer must take additional cards as long as its total is less than 17. If the computer busts, all players who have not themselves busted, win. Otherwise, each remaining player's total is compared with the computer's. If the player's total is greater, the player wins. If the player's total is less, the player loses. If the two totals are the same, the player ties the computer (also known as "pushing"). [Figure 9.1](#) shows off the game.



```
C:\Python22\python.exe
Welcome to Blackjack!
How many players? (1 - 7): 2
Enter player name: Larry
Enter player name: Jerry

Larry: 7h    7s    (14)
Jerry: 10d   9h    (19)
Dealer: Kh   3c

Larry, do you want a hit? (Y/N): y
Larry: 7h    7s    2d    (16)

Larry, do you want a hit? (Y/N): y
Larry: 7h    7s    2d    10c    (26)
Larry busts.
Larry loses.

Jerry, do you want a hit? (Y/N): n
Dealer: 10h   3c    (13)
Dealer: 10h   3c    Kh    (23)
Dealer busts.
Jerry wins.


Do you want to play again?:
```

Figure 9.1: One player wins, the other is not so lucky.

Back to the Blackjack Game

At this point, you're an expert in using Python classes to create playing cards, hands, and decks. So now it's time to build on that expertise and see how to combine these classes in a larger program to create a complete, casino-style, card game (tacky green felt not included).

The Cards Module

To write the Blackjack game, I created a final `cards` module based on the Playing Cards programs. The `Hand` and `Deck` classes are exactly the same as those in the Playing Cards 2.0 program. The new `Card` class represents the same functionality as the `Positionable_Card` from the Playing Cards 3.0 program. Here's the code for this module stored in the file  `cards.py` :

```
# Cards Module
# Basic classes for a game with playing cards
# Michael Dawson 4/18/03

class Card(object):
    """ A playing card. """
    RANKS = ["A", "2", "3", "4", "5", "6", "7",
             "8", "9", "10", "J", "Q", "K"]
    SUITS = ["c", "d", "h", "s"]

    def __init__(self, rank, suit, face_up = True):
        self.rank = rank
        self.suit = suit
        self.is_face_up = face_up

    def __str__(self):
        if self.is_face_up:
            rep = self.rank + self.suit
        else:
            rep = "XX"
        return rep

    def flip(self):
        self.is_face_up = not self.is_face_up

class Hand(object):
    """ A hand of playing cards. """
    def __init__(self):
        self.cards = []

    def __str__(self):
        if self.cards:
            rep = ""
            for card in self.cards:
                rep += str(card) + "\t"
        else:
            rep = ""
```

```

        rep = "<empty>"
    return rep

def clear(self):
    self.cards = []

def add(self, card):
    self.cards.append(card)

def give(self, card, other_hand):
    self.cards.remove(card)
    other_hand.add(card)

class Deck(Hand):
    """ A deck of playing cards. """
    def populate(self):
        for suit in Card.SUITS:
            for rank in Card.RANKS:
                self.add(Card(rank, suit))

    def shuffle(self):
        import random
        random.shuffle(self.cards)

    def deal(self, hands, per_hand = 1):
        for rounds in range(per_hand):
            for hand in hands:
                if self.cards:
                    top_card = self.cards[0]
                    self.give(top_card, hand)
                else:
                    print "Can't continue deal. Out of cards!"

if __name__ == "__main__":
    print "This is a module with classes for playing cards."
    raw_input("\n\nPress the enter key to exit.")

```

Designing the Classes

Before you start coding a project with multiple classes, it can help to map them out on paper. You might make a list and include a brief description of each class. Table 9.1 shows my first pass at such a listing for the Blackjack game.

BJ_Card

cards.Card

A blackjack playing card. Define an attribute `value` to represent the point value of a card.

BJ_Deck

cards.Deck

A blackjack deck. A collection of `BJ_Card` objects.

`BJ_Hand`
`cards.Hand`

A blackjack hand. Define an attribute `total` to represent the point total of a hand. Define an attribute `name` to represent the owner of the hand.

`BJ_Player`
`BJ_Hand`

A blackjack player.

`BJ_Dealer`
`BJ_Hand`

A blackjack dealer.

`BJ_Game`
`object`

A blackjack game. Define an attribute `deck` to reference a `BJ_Deck` object. Define an attribute `dealer` to reference a `BJ_Dealer` object. Define an attribute `players` to reference a list of `BJ_Player` objects.

Table 9.1: BLACKJACK CLASSES

Class	Base Class	Description
-------	------------	-------------

You should try to include all of the classes you think you'll need, but don't worry about making your class descriptions complete, because invariably they won't be (mine aren't). But making such a list should help you get a good overview of the types of objects you'll be working with in your project.

In addition to describing your classes in words, you might want to draw a family tree of sorts to visualize how your classes are related. That's what I did in Figure 9.8 .

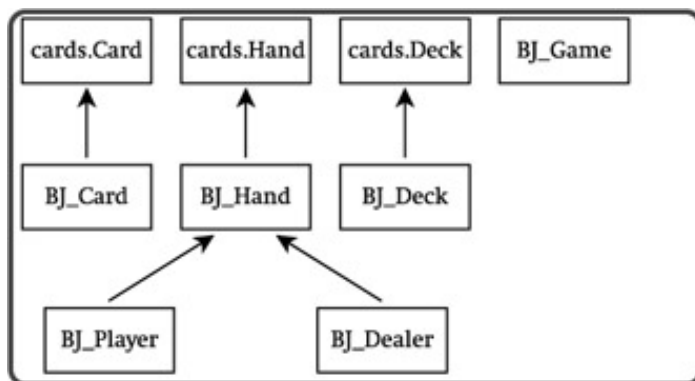


Figure 9.8: Inheritance hierarchy of classes for the Blackjack game.

A class hierarchy diagram, like the one in Figure 9.8 , can give you a summary view of how you're using inheritance.

Writing Pseudocode for the Game Loop

The next thing I did in planning the game was write some pseudocode for the play of one round. I thought this would help me see how objects will interact. Here's the pseudocode I came up with:

Deal each player and dealer initial 2 cards
For each player

```

    While the player asks for a hit and the player is not busted
    Deal the player an additional card
If there are no players still playing
    Show the dealer's 2 cards
Otherwise
    While the dealer must hit and the dealer is not busted
    Deal the dealer an additional card
    If the dealer is busted
    For each player who is still playing
    The player wins
Otherwise
    For each player who is still playing
    If the player's total is greater than the dealer's total
    The player wins
    Otherwise, if the player's total is less than the dealer's total
    The player loses
Otherwise
    The player pushes

```

Importing the cards and games Modules

Now that you've seen the planning, it's time to check out the code. In the first part of the Blackjack program, I import the two modules `cards` and `games` :

```

# Blackjack
# From 1 to 7 players compete against a dealer
# Michael Dawson 4/18/03

import cards, games

```

I created the `games` module, you'll remember, in the Simple Game program, earlier in this chapter.

The BJ_Card Class

The `BJ_Card` class extends the definition of what a card is by inheriting from `cards.Card` . In `BJ_Card` , I create a new property, `value` , for the point value of a card:

```

class BJ_Card(cards.Card):
    """ A Blackjack Card. """
    ACE_VALUE = 1

    def get_value(self):
        if self.is_face_up:
            value = BJ_Card.RANKS.index(self.rank) + 1
            if value > 10:
                value = 10
        else:
            value = None
        return value

```

```
value = property(get_value)
```

The `get_value()` method returns a number between 1 and 10, which represents the value of a blackjack card. The first part of the calculation is computed through the expression `BJ_Card.RANKS.index(self.rank) + 1`. This expression takes the `rank` attribute of an object (say "6") and finds its corresponding index number in `BJ_Card.RANKS` through the list method `index()` (for "6" this would be 5). Finally, 1 is added to the result since the computer starts counting at 0 (this makes the value calculated from "6" the correct 6). However, since `rank` attributes of "J", "Q", and "K" result in numbers larger than 10, any value greater than 10 is set to 10. If an object's `face_up` attribute is `False`, this whole process is avoided and a value of `None` is returned. Finally, I use the `property()` function with the `get_value()` method to create the property `value`.

The BJ_Deck Class

The `BJ_Deck` class is used to create a deck of blackjack cards. The class is almost exactly the same as its base class, `cards.Deck`. The only difference is that I override `cards.Deck`'s `populate()` method so that a new `BJ_Deck` object gets populated with `BJ_Card` objects:

```
class BJ_Deck(cards.Deck):
    """ A Blackjack Deck. """
    def populate(self):
        for suit in BJ_Card.SUITS:
            for rank in BJ_Card.RANKS:
                self.cards.append(BJ_Card(rank, suit))
```

The BJ_Hand Class

The `BJ_Hand` class, based on `cards.Hand`, is used for blackjack hands. I override the `cards.Hand` constructor and add a `name` attribute to represent the name of the hand owner:

```
class BJ_Hand(cards.Hand):
    """ A Blackjack Hand. """
    def __init__(self, name):
        super(BJ_Hand, self).__init__()
        self.name = name
```

Next, I override the inherited `__str__()` method to display the total point value of the hand:

```
def __str__(self):
    rep = self.name + ":\t" + super(BJ_Hand, self).__str__()
    if self.total:
        rep += "(" + str(self.total) + ")"
    return rep
```

I concatenate the object's `name` attribute with the string returned from the `cards.Hand` `__str__()` method for the object. Then, if the object's `total` property isn't `None`, I concatenate the string representation of the value of `total`. Finally, I return that string.

Next, I create a property called `total` , which represents the total point value of a blackjack hand. If a blackjack hand has a face-down card in it, then its `total` property is `None` . Otherwise, `total` is calculated by adding the point values of all the cards in the hand.

```
def get_total(self):
    # if a card in the hand has value of None, then total is None
    for card in self.cards:
        if not card.value:
            return None
    # add up card values, treat each Ace as 1
    total = 0
    for card in self.cards:
        total += card.value

    # determine if hand contains an Ace
    contains_ace = False
    for card in self.cards:
        if card.value == BJ_Card.ACE_VALUE:
            contains_ace = True

    # if hand contains Ace and total is low enough, treat Ace as 11
    if contains_ace and total <= 11:
        # add only 10 since we've already added 1 for the Ace
        total += 10

    return total

total = property(get_total)
```

The first part of this method checks to see if any card in the blackjack hand has a `value` attribute equal to `None` (which would mean that the card is face down). If so, the method returns `None` . The next part of the method simply sums the point values of all the cards in the hand. The next part determines if the hand contains an ace. If so, the last part of the method determines if the card's point value should be 11 or 1. The last line of this section creates the property `total` .

The last method in `BJ_Hand` is `is_busted()` . It returns `True` if the object's `total` property is greater than 21 . Otherwise, it returns `False` .

```
def is_busted(self):
    return self.total > 21
```

Notice that in this method, I return the result of the condition `self.total > 21` instead of assigning the result to a variable and then returning that variable. You can create this kind of `return` statement with any condition (any expression actually) and it often results in a more elegant method.

This kind of method, which returns either `True` or `False` , is pretty common. It's often used (like here) to represent a condition of an object with two possibilities, such as "on" or "off," for example. This type of method almost always has a name that starts with the word "is," as in `is_on()` .

The `BJ_Player` Class

The `BJ_Player` class, derived from `BJ_Hand` , is used for blackjack players:

```
class BJ_Player(BJ_Hand):
    """ A Blackjack Player. """
    def is_hitting(self):
        response = games.ask_yes_no("\n" + self.name + ", do you want a hit? (Y/N): ")
        return response == "y"

    def bust(self):
        print self.name, "busts."
        self.lose()

    def lose(self):
        print self.name, "loses."

    def win(self):
        print self.name, "wins."

    def push(self):
        print self.name, "pushes."
```

The first method, `is_hitting()` , returns `True` if the player wants another hit and returns `False` if the player doesn't. The `bust()` method announces that a player busts and invokes the object's `lose()` method. The `lose()` method announces that a player loses. The `win()` method announces that a player wins. And the `push()` method announces that a player pushes. The `bust()` , `lose()` , `win()` , and `push()` methods are so simple that you may wonder why they exist. I put them in the class because they form a great skeleton structure to handle the more complex issues that arise when players are allowed to bet (which they will, when you complete one of the chapter challenges at the end of the chapter).

The BJ_Dealer Class

The `BJ_Dealer` class, derived from `BJ_Hand` , is used for the game's blackjack dealer:

```
class BJ_Dealer(BJ_Hand):
    """ A Blackjack Dealer. """
    def is_hitting(self):
        return self.total < 17
    def bust(self):
        print self.name, "busts."

    def flip_first_card(self):
        first_card = self.cards[0]
        first_card.flip()
```

The first method, `is_hitting()` , represents whether or not the dealer is taking additional cards. Since a dealer must hit on any hand totaling 17 or less, the method returns `True` if the object's `total` property is less than 17, otherwise it returns `False` . The `bust()` method announces that the dealer busts. The `flip_first_card()` method turns over the dealer's first card.

The BJ_Game Class

The `BJ_Game` class is used to create a single object that represents a blackjack game. The class contains the code for the main game loop in its `play()` method. However, the mechanics of the game are complex enough that I created a few elements outside the method, including an `__additional_cards()` method that takes care of dealing additional cards to a player and a `still_playing` property that returns a list of all players still playing in the round.

The `__init__()` Method

The constructor receives a list of names and creates a player for each name. The method also creates a dealer and deck.

```
class BJ_Game(object):
    """ A Blackjack Game. """
    def __init__(self, names):
        self.players = []
        for name in names:
            player = BJ_Player(name)
            self.players.append(player)

        self.dealer = BJ_Dealer("Dealer")

        self.deck = BJ_Deck()
        self.deck.populate()
        self.deck.shuffle()
```

The `still_playing` Property

The `still_playing` property returns a list of all the players that are still playing (those that haven't busted this round):

```
def get_still_playing(self):
    remaining = []
    for player in self.players:
        if not player.is_busted():
            remaining.append(player)
    return remaining

# list of players still playing (not busted) this round
still_playing = property(get_still_playing)
```

The `__additional_cards()` Method

The `__additional_cards()` method deals additional cards to either a player or the dealer. The method receives an object into its `player` parameter, which can be either a `BJ_Player` or `BJ_Dealer` object. The method continues while the object's `is_busted()` method returns `False` and its `is_hitting()` method returns `True`. If the object's `is_busted()` method returns `True`, then the object's `bust()` method is invoked.

```

def __additional_cards(self, player):
    while not player.is_busted() and player.is_hitting():
        self.deck.deal([player])
        print player
    if player.is_busted():
        player.bust()

```

Polymorphism is at work here in two method calls. The `player.is_hitting()` method call works equally well whether `player` refers to a `BJ_Player` object or a `BJ_Dealer` object. The `__additional_cards()` method never has to know which type of object it's working with. The same is true in the line `player.bust()`. Since both classes, `BJ_Player` and `BJ_Dealer`, each defines its own `bust()` method, the line creates the desired result in either case.

The `play()` Method

The `play()` method is where the game loop is defined and bares a striking resemblance to the pseudocode I introduced earlier:

```

def play(self):
    # deal initial 2 cards to everyone
    self.deck.deal(self.players + [self.dealer], per_hand = 2)
    self.dealer.flip_first_card()    # hide dealer's first card
    for player in self.players:
        print player
    print self.dealer

    # deal additional cards to players
    for player in self.players:
        self.__additional_cards(player)

    self.dealer.flip_first_card()    # reveal dealer's first

    if not self.still_playing:
        # since all players have busted, just show the dealer's hand
        print self.dealer
    else:
        # deal additional cards to dealer
        print self.dealer
        self.__additional_cards(self.dealer)

    if self.dealer.is_busted():
        # everyone still playing wins
        for player in self.still_playing:
            player.win()
    else:
        # compare each player still playing to dealer
        for player in self.still_playing:
            if player.total > self.dealer.total:
                player.win()
            elif player.total < self.dealer.total:
                player.lose()

```

```

        else:
            player.push()
    # remove everyone's cards
    for player in self.players:
        player.clear()
    self.dealer.clear()

```

Each player and dealer is dealt the initial two cards. The dealer's first card is flipped to hide its value. Next, all of the hands are displayed. Then, each player is given cards as long as the player requests additional cards and hasn't busted. If all players have busted, the dealer's first card is flipped and the dealer's hand is printed. Otherwise, play continues. The dealer gets cards as long as the dealer's hand total is less than 17. If the dealer busts, all remaining players win. Otherwise, each remaining player's hand is compared with the dealer's. If the player's total is greater than the dealer's, the player wins. If the player's total is less, the player loses. If the two totals are equal, the player pushes.

The `main()` Function

The `main()` function gets the names of all the players, puts them in a list, and creates a `BJ_Game` object, using the list as an argument. Next, the function invokes the object's `play()` method and will continue to do so until the player no longer wants to play.

```

def main():
    print "\t\tWelcome to Blackjack!\n"

    names = []
    number = games.ask_number("How many players? (1 - 7): ", low = 1, high = 8)
    for i in range(number):
        name = raw_input("Enter player name: ")
        names.append(name)
    print

    game = BJ_Game(names)

    again = None
    while again != "n":
        game.play()
        again = games.ask_yes_no("\nDo you want to play again?: ")

main()
raw_input("\n\nPress the enter key to exit.")

```

Summary

This chapter introduced you to the world of OOP. You saw how to send messages between objects. You learned how to combine objects together to form more complex objects. You were introduced to inheritance, the process of creating new classes based on existing ones. You saw how to extend a derived class by adding new methods. You also saw how to override inherited methods. You learned how to write and import your own modules. You were shown an example of how to sketch out your classes before you begin a project. And finally, you saw all of these concepts come together in the creation of a multiplayer, casino-style card game.

Challenges

1. Add some much-needed error checking to the Blackjack game. Before a new round begins, make sure that the deck has enough cards. If not, repopulate and reshuffle it. Find other places where you could add error checking and create the necessary safeguards.
2. Write a one-card version of the game war, where each player gets a single card and the player with the highest card wins.
3. Improve the Blackjack project by allowing players to bet. Keep track of each player's bankroll and remove any player who runs out of money.
4. Create a simple adventure game, using objects, where a player can travel between various, connected locations.