Object-Oriented Programming, Part 4 of 4

Christopher Simpkins
chris.simpkins@gatech.edu

Tying it All Together

Toays we'll employ all the things we've learned so far in the design of a Blackjack card game.

- We'll reuse the `PlayingCard` class we wrote a few weeks ago
- The only new langauge feature we'll learn is interfaces
- We'll practice testing, debugging, and bottom-up programming

Blackjack

Let's use these rules:

- Two players: "the house" and human player
- House is played by computer
- Player is played by a human interacting with the program
- Game begins by dealing each player a face down and face up card.
- Gameplay proceeds in rounds during which
  - Players can "hit", meaning draw another card face up
  - Players can "stand" meaning keep current hand
- When human player stands, game ends

Note that I'm using the term "house" instead of "dealer" to avoid confusion when we talk about dealing cards, etc.

Blackjack Classes

We'll need the following classes:

- PlayingCard
- Deck (of cards)
- BlackjackHand
- Player (actually, a hierarchy of Player classes)
- BlackjackGame - to run the game

Notice that these are the significant nouns from the problem description. In general, look for nouns to help you identify which classes you'll need to write in an OO program.

Now let's start hacking!

Enum Types

Enums are data types that have a predefined set of constant values (JLS §8.9, Java Enum Tutorial)
For example:

```
public enum Suit {DIAMONDS, CLUBS, HEARTS, SPADES};
```

defines an enum type called `Suit` that can take on only one of the predefined constants `Suit.DIAMONDS`, `Suit.CLUBS`, `Suit.HEARTS`, or`Suit.SPADES`

- Enum types are a class.
- Java automatically defines convenience methods for enum types, like `valueOf` and `values`.
- Because they define a class, enum types can include programmer-defined additional constructors and methods.

Take a look at PlayingCard.java for an example.

Deck

With a Deck we should be able to

- contain all 52 standard playing cards
- *shuffle* the cards in the deck
- allow user to *draw* a card off the top of the deck

In the description of a class's responsibilities, look for significant verbs to help you identify which methods need to be part of the class's public interface.

BlackjackHand

With a BlackjackHand we should be able to

- *add a card face-up*
- *add a card face-down*
- get the *value* of the hand
- *compareTo* other BlackjackHands

Comparing `BlackjackHand`s

To permit comparison of BlackjackHands we'll implement the `java.lang.Comparable` interface.

- An interface is like an abstract class with nothing but abstract methods
- A class can only `extend` one superclass, but can `implements` any number of interfaces
- An interface defines a type by specifying its API without specifying any of its implementation
- In OO parlance a type is the set of messages an object can respond to, or the public methods that an object implements

Implementing interfaces is as straightforward as extending abstract classes. Let's implement the `Comparable` interface in our `BlackjackHand` class.

The `java.lang.Comparable` Interface

```
public interface Comparable<T> {

    public int compareTo(T o);
}
```

And, of course, the API documentation: `http://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html`

Player

A player

- holds a deck of cards
- makes moves, e.g., to draw or stand
- may be a computer-controlled player or a human player

Player objects will be central to the logic of our game, that is, how the game runs turn by turn. So before we write our Player classes, let's write the game itself so we know what we need from Player objects.

BlackjackGame

The general form of a Blackjack game looks something like this:

1. Shuffle the deck.
2. Deal each player a face-down card
3. Deal each player a face-up card
4. While the human player keeps drawing
    1. Get the player's move
    2. Execute the player's move
    3. Get the house's move
    4. Execute the house's move
5. Evaluate each players' hand and declare a winner

Let's write our BlackjackGame class and then go bak and write our Player class hierarchy.

Fin! Lessons Learned

- Look for nouns in problem descriptions to determine which classes you'll need in an OO program.
- Look for verbs to determine which methods should be a part of a class's public interface.
- Write programs "bottom-up", that is, in small chunks, testing each small chunk as you go.
- Use main methods in classes to test classes in isolation.
- Interfaces define an OO type (public methods) without defining any implementation at all.
- Implementing an interface is an example of *interface* inheritance, as opposed to the *implementation* inheritance we've done when extending classes.
- Writing the client code of a class before you write the class itself is a good way to determine what the public API of a class should be. This idea is also a central concept in test-driven development.