CSCI 136: Fall 2019
Handout 5
25 September

Java Interfaces

Interfaces in Java

Let's use our `BasicCard` class example to explore more of the features supported by the Java class construct.

We chose to represent a card with two `enum` values, one each for the rank and the suit. We could have decided to represent a card using two integers instead: a rank in the range $\{0, \ldots, 12\}$ (or $\{1, \ldots, 13\}$, or $\{2, \ldots, 14\}$, etc) and a suit in the range $\{0, \ldots, 3\}$ (or $\{1, \ldots, 4\}$, ...). Or we could have chosen to represent each card by a number in the range $\{0, \ldots, 51\}$. Each of these representations has advantages and disadvantages; in different situations we might choose one over another[*].

How can we design our card code so that we could support multiple implementations—and do so with a minimal amount of code duplication? The first step is to make a distinction between *what* functionality cards provide and *how* they provide it; that is, to separate the *interface* of the card class from its *implementation(s)*. Before doing this, we'll make a small, but useful, digression.

We said earlier[†] that Java provides four categories of types: primitive, class, array, and enum types (and later admitted that enum wasn't really a separate category, just a special kind of class type). Class, array, and enum types are referred to as *reference types*: a variable of one of these types, does not hold an actual object (or entire array) but rather holds a reference to (information about the location in memory of) the object or array. When you assign a value to a variable of some primitive type, the variable stores that value; when you assign a value to a class (or array) type variable, the variable stores a reference to to the actual object (or array).

This difference has several important implications. Suppose I execute the following lines of Java code:

```
int x = 3, y=0;
y = x;
x++;
```

Now `x` has the value 4, but `y` still has the value 3: the statement `y = x` *copied* the value of `x` to `y`. Subsequent changes to `x` do not affect `y`. Now consider the following code fragment:

```
Student a = new Student(18, "Patti Smith", 'A');
Student b = new Student(20, "Joan Jett", 'B');
b = a;
a.setAge(19);
```

What is the age associated with student `b`? It is 19 because the statement `b = a` copied the *reference* to the student named Patti Smith from variable `a` to variable `b`. Thus `a` and `b` now reference the exact same `Student` object, not separate copies of it (and poor Joan Jett has been lost forever). Thus any change to the object referenced by `a` is also a change to the (same) object referenced by `b`[‡].

So, all of the types described so far are types that let us create actual values, of either primitive or reference type. But there is another variety of type that does not let us create actual values. It is called an *interface type*. An interface lets us specify functionality without providing an implementation. It can include the declaration of constants and method signatures (but not method implementations). Once an interface has been written, other classes can *implement* that interface: that is, they can provide implementations of the method the interface describes. Let's consider our card example.

[*] Although, admittedly, in an example this simple, the stakes are pretty low; in the coming weeks we'll see that data structure selection often has dramatic performance implications.

[†] Java Essentials handout

[‡] This same reasoning applies to values passed as parameters to, and returned from, methods.

We replace the `Card` class with an interface:

```
public interface Card {

        // Methods - must be public
        public Suit getSuit();
        public Rank getRank();
}
```

The declarations of the `Rank` and `Suit` enums have been moved to their own files; the `Suit` enum looks like:

```
public enum Suit {
        CLUBS, DIAMONDS, HEARTS, SPADES; // the values

        public String toString() {
                switch (this) {
                        case CLUBS : return "clubs";
                        case DIAMONDS : return "diamonds";
                        case HEARTS : return "hearts";
                        case SPADES : return "spades";
                }
                return "Bad suit!";
        }

        public static void main(String[] args) {
        for( Suit s : Suit.values()) System.out.println(s);
        }
}
```

The `Suit` (and `Rank`) enums provide a `toString` method to provide a nice ”no caps” representation of their values. Note that the Card interface has no instance variables or executable code, just a description of the methods that any class claiming to implement the interface should provide. Also note that there are no constructors—unlike a class, an interface cannot create (instantiate) objects itself and there are no values to initialize.

Here’s a class, `CardRankSuit` that implements the `Card` interface.

```
public class CardRankSuit implements Card
{
    // "protected" means other classes can't access them
    //  (data hiding)

    // instance variables
    protected Suit suit;     // The suit of card: CLUBS..SPADES
    protected Rank rank;     // The rank of the card: TWO..ACE

    // Constructors

    // Constructs a card of the given type
    public CardRankSuit( Rank theRank, Suit theSuit) {
        suit = theSuit;
        rank = theRank;
    }

    // returns suit of card
    public Suit getSuit() {
        return suit;
    }

    // returns rank of card
    public Rank getRank() {
        return rank;
    }

    public String toString() {
        return getRank() + " of " + getSuit();
     }

    public static void main(String s[]) {
        Card ace = new CardRankSuit( Rank.ACE, Suit.SPADES );
        Card three = new CardRankSuit( Rank.THREE, Suit.DIAMONDS );

        System.out.println(ace);
        System.out.println(three);

    }

}
```

Note that the first line of the class declaration above explicit states that CardRankSuit *implements* `Card`. This imposes the requirement that each method in the `Card` interface be implemented by `CardRankSuit`. The methods that are declared in `Card` must have exactly the same signature in `CardRankSuit` as they do in `Card`. Also note that in the method `main`—a modest test of the class `CardRankSuit`— two variables are *declared* to be of type `Card` but are *instantiated* (created) as type `CardRankSuit`. *Any* class that implements `Card` can be used instead of `CardRankSuit`. One very useful consequence of this is that one can write methods with parameters of type `Card` and invoke them with any values from classes that implement `Card`.

This allows us to write code that manipulates objects but that is independent of the actual implementation of those objects! For example, we could just as easily have developed a much different implementation[§] of the `Card` type:

```
public class Card52 implements Card
{
    // "protected" means other classes can't access them
    //  (data hiding)

    // instance variables
    protected int code; // 0 <= code < 52; suit = code / 13; rank = code % 13

    // Constructors

    // Constructs a card of the given type
    public Card52( Rank theRank, Suit theSuit) {
        code = theSuit.ordinal() * 13 + theRank.ordinal();
    }

    public Card52( int index) {
        code = index;
    }

    // returns suit of card
```

[§]Yet another implementation is provided on the course web site.

```
    public Suit getSuit() {
        return Suit.value( code / 13 );
    }

    // returns rank of card
    public Rank getRank() {
        return Rank.value( code % 13 );
    }

    public String toString() {
        return getRank() + " of " + getSuit();
    }

    public static void main(String s[]) {
        Card ace = new Card52( Rank.ACE, Suit.SPADES );
        Card three = new Card52(14);

        System.out.println(ace);
        System.out.println(three);
    }
}
```

Here each card is encoded by a single integer and the class internally computes the appropriate rank and suit values as needed. In particular, note that `getRank` and `getSuit` are quite different from their counterparts in the class `CardRankSuit`—all that is necessary is that they produce the correct value. Also note that two constructors were provided since it seems reasonable that to provide a `Rank/Suit` version as well as an index version—only one of the two is needed, and the `Rank/Suit` version is arguably preferable.