Java Class Types

Class Types

We noted earlier that the `String` type in Java was *not* a primitive (or array) type. It is what Java calls a *class-based* (or *class*) type—this is the third category of types in Java. Class types, are based on *class declarations*. Class declarations allow us to create *objects* that can hold more complex collections of data, along with operations for accessing/modifying that data. For example

```
public class Student {
        private int age;
        private String name;
        private char grade;

        public Student(int theAge, String theName, char theGrade) {
                age = theAge;
                name = theName;
                grade = theGrade;
        }

        public int getAge() { return age;}

        public String getName() { return name;}

        public char getGrade() { return grade;}

        public void setAge(int newAge) { age = newAge;}

        public void setGrade(char grade) { this.grade = grade;}
}
```

The above, grotesquely oversimplified, class declaration specifies the data components (*instance variables*) of a *Student* object, along with a method (called a *constructor*[*]) describing how to create such objects, and several short methods for accessing/modifying the data components (*fields/instance variables*) of a `Student` object. Given such a class declaration, one can write programs that declare (and create) variables of type Student:

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

Or, combining the declaration of the variables and creation (*instantiation*) of the corresponding values (objects):

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

The words `public` and `private` are called *access level modifiers*; they control the extent to which other classes can create, access, or modify objects, their fields, and their methods. Declaring the class `Student` to be `public` allows other classes to create objects of type `Student`. Likewise, declaring the methods `getName, setGrade`, etc., to

[*] Note: The constructor uses the same name as the class

be `public` allows other classes to invoke these methods on any `Student` objects they have created. Declaring the instance variables (`name, age, grade`) to be `private` blocks other classes from directly accessing or modifying those variables. An almost universal rule of thumb in object-oriented design is to allow instance variables to be accessed/modified *only* through the use of class methods. This helps guarantee that the underlying state of an object can't be compromised by users and that the underlying implementation of a class can be changed if needed without compromising the functionality of programs that use the class.

The ability to create new data types through class declarations allows for the construction of robust and reusable code modules and supports the development of larger bodies of code. Class types can be used very much like primitive types: Variables of any class type can be created, passed to (and returned from) methods, used as types of instance variables for even more complex classes, and so on. One can create arrays of variables of any class type:

```
// Create an array to store 3 objects of type Student
Student[] class = new Student[3];

// Create the three Student objects and store them in the array
class[0] = new Student(18, "Patti Smith", 'A');
class[1] = new Student(20, "Joan Jett", 'B');
class[2] = new Student(20, "David Bowie", 'A');
```

Note the use of `new` here, both for the creation of the individual student objects and for the creation of the array. Array and class-based types have more complex *storage requirements* for their values than do primitive types; in Java, that storage is allocated by using the keyword `new` followed by an invocation of the array or class constructor. The two exceptions to this norm are

- The allocation of an array by explicitly listing its values: `int[] scores = {97, 85, 100};`,
- The creation of a String using a String *literal* [†]: `String name = "Zeta";`.

Strings are unique in Java among class-based types in that String values can be specified by String literals; the only other types whose values can be specified by literals are the primitive types.
Strings and arrays in Java have a very similar flavor, but there are some key differences; among them are:

- The i^{th} element of an array x is referenced with syntax $x[i]$; the i^{th} character in a String x is referenced with syntax $x.substring(i, i+1)$.
- Strings are *immutable*: one cannot assign a value to an individual position in a String variable; rather a new String can be constructed by piecing together (*concatenating*) other Strings.
- To get the size (length) of a String x, use $x.length()$ (i.e., invoke the `length` method of the `String` class); to get the size of an array x, use $x.length$ (i.e., access the `length` instance variable of the array x).

The Structure of a Java Program

A Java program consists of a set of class declarations; each class declaration typically describes a type of object that can be created and includes any object data (*instance variables*) and functionality (*class methods*). An executing program consists of a sequence of statements that declare and construct objects and then invoke the methods of the objects in order to access or modify them in some way. These statements are woven together with other statements that control the flow of program execution ("if" statements, looping constructs, and so on). Java itself is not a large language; the set of keywords and symbols in the language is modest. What makes the language powerful and flexible is the ability to add functionality by designing new class types.
Java is designed to be run in many different environments, from stand-alone code on a computer to embedded systems on a wide range of devices. The method for executing Java code that we will focus on is the use of a special method, (always) named *main* that we can include in a Java class declaration. Here's a simple Java program:

[†]A literal is an explicit representation of a value in Java source code, such as `21, 3.14159, 'C', true, "Hi there!"`. The only other literal for class types is `null`, which can be assigned to any variable of non-primitive type.

```
import Student;

public class StudentDemo {

        pubic static void main(String[] args) {
                Student a = new Student(18, "Patti Smith", 'A');
                Student b = new Student(20, "Joan Jett", 'B');

                if( a.grade() == b.grade() )
                        System.out.println("Grades match");
                else
                        System.out.println("Grades don't match");
        }
}
```

The program above consists entirely of a `main` method. The method itself is pretty dull, it merely compares the grades of the two student objects and prints an appropriate message. The first line of the `main` method, called the method *signature*, always has the form `public static void main(String [] args)`[‡]. We'll talk more about the meanings of the keywords *public, static, void*, but, essentially, they indicate that

public the method can be invoked by users of the Student class

static the method can be invoked (called) without reference to a particular object of type Student; that is, the method can be called with the syntax: `Student.main(x)`, where `x` is an array of Strings

void the method does not return a value

Using any text editor, we can create a file that contains the class declaration above, giving the file the name `StudentDemo.java` (always use the name of the class as the name of the file). We *compile* the program (convert it into Java bytecode) by typing `javac StudentDemo.java` in a terminal window. We can then execute the bytecode of the `main` method of the program by typing `java StudentDemo`.
The `StudentDemo` class also includes an `import` statement. This statement ensures that the `Student` class is available for use in the `StudentDemo` class. Here's another example of a class that It's worth noting that we can define a class that consists *only* of a `main` method. For example,

```
public class MathCalcs {

        import java.lang.Math;

        pubic static void main(String[] args) {
                double x = Math.pow(E,PI); // e^pi
                double y = Math.pow(PI,E)/ // pi^e
                if (x > y)      System.out.println("e^pi is greater than pi^e");
                else if (y > x) System.out.println("pi^e is greater than e^pi");
                else            System.out.println("e^pi equals pi^e");
        }
}
```

Note that we import the `Math` class from the `java.lang` *package*. A package is a collection of classes that have been bundled together to provide a family of services. The `java.lang` package is part of the standard Java distribution.

Enumeration Types

Before exploring the properties and uses of class types, let's briefly look at a fourth category of types provided by Java: *enumeration types*. Enumeration types are used to provide families of *named constants*. For example, we could create named constants for the four suits and 13 ranks of a deck of playing cards as follows;

[‡]Well, the name `args` can be replaced by any other legal variable name....

```
public enum Suits { CLUBS, DIAMONDS, HEARTS, SPADES }

public enum Ranks {TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN,
                JACK, QUEEN, KING, ACE }
```

We could then use these names to create a `Card` class:

```
public class Card {

        private Suit s;
        private Rank r;

        public Card(Suit s, Rank r) {
                this.s = s;
                this.r = r;
        }

        public Suit getSuit() { return s; }
        public Rank getRank() { return r; }
        public void setSuit(Suit s) { this.s = s; }
        public void setRank(Rank r) { this.r = r; }

}
```

This code could then be used to create a deck of cards:

```
Card[] deck = new Card[52];

int i = 0;
for ( Suit s : Suit.values() )
        for ( Rank r : Rank.values() ) {
                Card[i] = new Card( s, r );
                i++;
```

The file BasicCard.java contains an implementation of a simple card class like the one above. Note that the keyword `public` is missing from the two `enum` declarations. This is because a single file can only contain one public class declaration. Which leads us to note that calling enumeration types a "fourth category" is somewhat misleading. An enumeration type is just a special kind of class type that allows us to define classes having fixed numbers of possible values and that can be iterated over using the `for-each` loop construct in Java. In the following more thorough versions of our playing card example, we put each enum declaration in its own file, where we can declare them `public`. More on this later.

Classes: Interfaces, Implementation, and Abstraction

Let's use our `Card` 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 $\{w, \ldots, 14\}$, 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[§].

[§] 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.

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.

We replace the `Card` class with an interface:

```
public interface Card {

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

[¶] Java Essentials handout

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

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 Card52v2 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 Card52v2( Rank theRank, Suit theSuit) {
        code = theSuit.ordinal() * 13 + theRank.ordinal();
    }

    public Card52v2( int index) {
        code = index;
    }

    // returns suit of card
    public Suit getSuit() {
        return Suit.value( code / 13 );
    }

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

** Yet another implementation is provided on the course web site.

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

    public static void main(String s[]) {
        Card ace = new Card52v2( Rank.ACE, Suit.SPADES );
        Card three = new Card52v2(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.

While the `getRank` and `getSuit` methods are different for each class that implements `Card` notice that the `toString` method is identical; in this case, because it only uses methods defined in the interface. A good coding practice is to always attempt to *factor out* common code and put it in one place; this avoids having to maintain the same code in multiple places. Java provides a way to do this: *abstract classes*.

An abstract class is merely one which is declared with the Java keyword `abstract`. Like an interface, an abstract class cannot instantiate objects, but unlike an interface, an abstract class can have its own instance variables, and implement methods. These features make abstract classes good intermediates between interfaces and complete (non-abstract) class declarations: Any method that has the same implementation across all classes that implement the interface can be put in the abstract class[††].

Below we show an abstract base class for our card example, and describe how its presence changes the classes that fully implement `Card`.

```
public abstract class CardAbstract implements Card
{
    public String toString() {
        return getRank() + " of " + getSuit();
    }
}
```

We want to position this class "between" the `Card` interface and the full implementations: `CardRankSuit, Card52`, etc.. We do this by indicating that `CardAbstract` implements `Card`[‡‡]. The only method that can be factored out in this example is the `toString` method, so we include it here and remove it from the individual implementing classes. We also need to indicate that the implementing classes `CardRankSuit, Card52`, etc. know about and can use the code in `CardAbstract`. We do this by saying that these classes *extend* `CardAbstract`. We no longer have to state that `CardRankSuit, Card52`, etc. `implement Card`; because they are extending a class that implements `Card`, they themselves automatically are classes that implement `Card`. Other than removing the `toString` method and altering the "class" statement in each of `CardRankSuit, Card52`, etc., no further modifications are needed; the code for these classes is available on the course web site.

This decomposition of our code into an interface, one (or more) abstract classes, and one or more fully implemented classes will appear repeatedly throughout the semester as we design our data structures.

[††] A class playing this role is sometimes called an *abstract base class*.

[‡‡] Clearly it only partially implements `Card`.

Extending Non-Abstract Classes

Continuing our playing card example, supposed we decided that for some applications we wanted each of our cards to be able to store a point value. How might we take advantage of our previous coding work? Java allow us to extend classes, adding new instance variables and methods. Here is an implementation of a class `CardRankSuitPoints` that extends `CardRankSuit` so that each card now has a point value.

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

    // instance variables
    protected int points;

    // Constructors

    // Constructs a card of the given type
    public CardRankSuitPoints( Rank theRank, Suit theSuit,
                               int pointVal) {
        super(theRank, theSuit);
        points = pointVal;
    }

    // Constructs a card of the given type
    public CardRankSuitPoints( Rank theRank, Suit theSuit ) {
        super(theRank, theSuit);
        // Default point value is "face" value
        points = 2 + theRank.ordinal();
    }

    // returns point value of card
    public int getPoints() {
        return points;
    }

    // Note: Probably don't want to add point value
    // to String representation of card, but it's done
    // to illustrate the overriding of a method
    public String toString() {
        return super.toString() + " (" + points + " points)";
    }

    public static void main(String s[]) {
        Card ace = new CardRankSuitPoints( Rank.ACE, Suit.SPADES, 20 );
        Card three = new CardRankSuitPoints( Rank.THREE, Suit.DIAMONDS );

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

Here are the salient features of this code:

- The phrase `extends CardRankSuit` is included in the `class` statement,
- A new protected instance variable `points` is added,

- The features of `CardRankSuit`—instance variables, methods—are *inherited* from `CardRankSuit` and so do not need to be rewritten here (unless it is desired to change their behavior),
- The constructor is modified to allow a parameter to pass in a point value for a card; that constructor, through the keyword `super` begins by calling the constructor for `CardRankSuit`,
- A second constructor allows for a default point setting, so no point value parameter needed,
- A new method `getPoints` is added, and the method `toString` is *overridden* so that when an object of type `CardRankSuitPoints` is converted to a `String`, the point value is included.

Hopefully this extended example has given you a sense of how combining interfaces, inheritance, and abstract classes provides support for flexible and efficient development of modular and reusable code. Before we end, let's put the code to use by creating a rudimentary deck of cards class. An object created by the `CardDeck` class provides a deck of 52 standard playing cards. When created, the deck is sorted from the 2 of clubs to the ace of spades. There is also a method, `shuffle` that randomly permutes the order of the cards in the deck, as well as a `toString` method. Finally, the method `main` creates a deck of cards, prints it out, then repeatedly shuffles it until an ace appears as the "top" card on the deck, reporting how many shuffles were needed and printing the deck in order.

```
public class CardDeck {

    static protected final int NUM_CARDS = 52;

    protected Card[] cards;
    protected Random gen;
    /*
     * Create a new random deck
     */
    public CardDeck() {
        // allocate array and create random number generator
        cards = new Card[NUM_CARDS];
        gen = new Random();

        int count = 0;
        for( Suit s : Suit.values() )
            for( Rank r : Rank.values() ) {
                cards[count] = new Card413( r, s );
                count++;
            }
        shuffle();
    }

   public void shuffle() {

        for (int remaining = cards.length; remaining > 1; remaining--) {
            int i = gen.nextInt(remaining);
            Card toMove = cards[i];
            cards[i] = cards[remaining-1];
            cards[remaining-1] = toMove;
        }
    }

    /*
     * Returns a string representation of the deck
     */
    public String toString() {
```

```
        String result = "";
        for (int i = 0; i < cards.length; i++) {
            result = result + cards[i] + "\n";
        }
        return result;
    }

    /*
     * Return true when top card is an ace
     */
    protected boolean isAceOnTop() {
        return Rank.ACE == cards[0].getRank();
    }

    public static void main(String s[]) {
        CardDeck deck = new CardDeck();
        System.out.println();
        System.out.println(deck);

        int count = 0;
        while(!deck.isAceOnTop()) {
            System.out.println("Not yet...");
            count++;
            deck.shuffle();
        }
        System.out.println("Deck #:" + count + " has an ace on top!");
        System.out.println(deck);
    }
}
```

Aspects of `CardDeck` worth noting

- The deck is held as an array of `Card`s; only when each card is created do we need to commit to a particular implementation,
- The shuffle method is cute: It picks a random position from 0 to 51 and swaps the card in that position with the card in position 51, then it picks a random position from 0 to 50 and swaps the card in that position with the card in position 50, and so forth. This is much more efficient than, say, generating new cards at random, while making sure that the same card is not generated more than once!

Some Final Notes: Testing Object Equality and the Object Class

Consider the following code fragment:

```
        Card a = new Card(Rank.KING, Suit.DIAMONDS);
        Card b = new Card(Rank.KING, Suit.DIAMONDS);
        Card c = a;
        System.out.println(a == c);
        System.out.println(a == b);
```

What does this code print? Not surprisingly, the first `println` will produce `true`; the second, however, will produce `false`. Why? While cards `a` and `b` both represent the king of diamonds, they are two different objects and the equality operator, when applied to class types, is checking equality of the *references*, not the actual contents of the objects. But we would often like to be able to determine whether two different objects of the same type have the same value (e.g., whether two different card objects both represent the king of diamonds). Java provides a method for this: the `equals` method. Adding the `equals` method to, say, `CardRankSuit` as illustrated below, allows us to test for the kind of equality (equivalent values in different cards) that we desire

```
public boolean equals(Object other) {
        Card c = (Card) other;
        return c.getRank() == this.rank && c.getSuit() == this.suit;
}
```

Now we can check equality of our cards by writing, say,

```
        if ( myCard.equals(yourCard) ) \{ ... \}
```

In fact, if we rewrote the code as follows

```
public boolean equals(Object other) {
        Card c = (Card) other;
        return c.getRank() == this.getRank() && c.getSuit() == this.getSuit();
}
```

we could then move the code into the abstract base class `CardAbstract` and wouldn't need to include it in all of the separate classes that implement the `Card` interface.

There are a number of situations in which some Java library class, for example the Java Collections classes, will use the `equals` method when comparing objects, so it is a good idea to add this method to any classes you write for which reference equality is *not* the appropriate version of equality for your class.

You might wonder why we made the type of the parameter `other` be `Object` rather than `Card`. The reason is the following. All class types in Java extend by default the class `Object`; it is the simplest reference type. The class `Object` includes a handful of methods, among them `toString` and `equals` that are therefore inherited by all class types. Because these methods have to work for every class-based type and yet still have the same method signature, the parameter to `equals` must be of type `Object`. In order to use the `Card` methods `getRank` and `getSuit` on the parameter, it must be cast from type `Object` to type `Card`.

What would happen if someone passed some object that was not of type `Card` to the `equals` method, say `myCard.equals(someStudent)`? The attempt to cast `other` of type `Student` to type `Card` would produce a run-time error. Java is a *type-safe* language and will only permit the casting of a value to its actual type or of a type that it extends. To avoid the run-time error, we could modify `equals` as follows:

```
public boolean equals(Object other) {
        if( other instanceof Card ) {
                Card c = (Card) other;
                return c.getRank() == this.getRank() &&
                        c.getSuit() == this.getSuit();
        }
        else return false;
}
```

Clearly if other is not even a `Card`, it can't be equal to a value that is a `Card`!