13. Enumerated Types

***Note**: some example programs are from, or based on, examples from Java for Everyone (C Horstmann), the course text.*

If you look back at the first chapter on Classes we had to deal with an issue:

```
public void setNameValue(String cardName){
     name = cardName; //sets the name
     switch(value) {
     case “Ace”:    value = 1;
               break;
     case “Two”:    value = 2;
               break;
     //others skipped to make it shorter
     case “King”:   value = 10;
               break;
     default:  value = -1; //An error value
               name = “error”;
               break;
     }
}
```

If the string `cardName` doesn’t match an expected value, we have to set it to some error value. This is a bit annoying – and we’d prefer not to deal with it.

It’s always better to stop incorrect data being entered in the first place than to check it afterwards.

(This isn’t only true in programming – for example, consider websites where you have to enter data. Drop-down lists which only let you choose the legal values are more convenient than text boxes where you could type anything which could well be wrong.)

Enumerated Types

What we want is some way of listing *only* the allowed values. Many programming languages have the concept of an *enumerated type*, including Java. An enumerated type lists, or *enumerates*, the permitted values. We’ve already seen a sort-of example of this – the `boolean` type just consists of two values – `true` and `false`.
Enumerated types aren’t suitable for everything – there need to be few enough values that you can reasonable list them all in a program (so a few tens at most).
Enumerated types have a very simple, basic, syntax in Java:

```
public enum CitrusFruit {
     LEMON, ORANGE, LIME, GRAPEFRUIT, KUMQUAT
}
```

(We said ‘basic’ syntax because we can do more complicated things and will do later on.)

We can use them quite simply:

```
CitrusFruit someFruit; //make a new CitrusFruit variable
someFruit = CitrusFruit.LEMON; //sets someFruit to LEMON
```

The only values a `CitrusFruit` variable can take are those listed in the definition of the type.

(Hopefully you can see now how this relates to cards – suites only have four possible values; and cards only 13 – well within the sensible sizes of possible values that make sense for an enumerated type.)

Enumerated Types in the Card Classes

Based on the example above, we can do something like this:

```
public enum Suite {
     //This a *new type* with only these values allowed
     CLUBS, DIAMONDS, HEARTS, SPADES
}
```

and it also looks as if we can do the same for the names – and we can, but we can also do a bit more than that. Recall that the name and value of a card are linked: once you know the name, you can work out the value. Can we include that somehow? Yes we can – but first we need to look at what enumerated types in Java *really* are.

Enumerated Types are Classes

Enumerated types are just a kind of renamed class – what’s actually happening, internally, hidden from us, is:

```
public class Suite {
     public static CLUBS = new Suite();
     public static DIAMONDS = new Suite();
     public static HEARTS = new Suite();
     public static SPADES = new Suite();

     private Suite() {
     }
}
```

We have created a *normal* class but unlike our usual approach where the variables are private and the methods and constructors are public (or at least some of them), here we have *public variables* and a *private constructor*. This means:

- The values of `CLUBS`, `DIAMONDS` etc. are visible outside the class (which we need to happen so we can use them).
- There can never be anymore `Suite` items than those we create inside `Suite` – because the constructor is private we can only call it from inside `Suite`, and we only do that four times.

Enumerated Types that Do More

So, if enumerated types are really classes, can we treat them like classes and add more functions? Yes – here's an enumerated type that handles names and values:

```
public enum CardName {
      ACE(1),          // These enum items are also
      TWO(2),          // constructor calls
      THREE(3),        // to the constructor CardName below
      FOUR(4),
      FIVE(5),
      SIX(6),
      SEVEN(7),
      EIGHT(8),
      NINE(9),
      TEN(10),
      JACK(10),
      QUEEN(10),
      KING(10);

      private final int value;

      // As well as creating the CardName Objects
      // ACE ... KING, they also add a *value*
      // to each one
      CardName(int cardValue){
            value = cardValue;
      }

      public int value() {
            return value;
      }
}
```

This time when we list the elements of the enumerated type, we pass an integer as a parameter. We've also explicitly written out the constructor for it, which saves that integer in a variable. We also add a method to return the value of that integer. (Note this time we've listed all the enumerated type values one per line, and before we didn't – there's no real 'rule' about this, provided it's readable.)

Having defined two enumerated types, we can now use them to redefine our Card class:

```
public class Card {

	private final Suite suite;
	private final CardName name;

	public Card(Suite suiteVal, CardName cardVal){
		name = cardVal;
		suite = suiteVal;
	}

	public Suite getSuite() {
		return suite;
	}

	public CardName getName() {
		return name;
	}

	public int getValue() {
		return name.value();
	}

	public String toString() {
		return (name + " of " + suite);
	}
}
```

- Now we don't pass in Strings as parameters, but variables of our two enumerated types.
- We don't do any error handling because *only valid values are possible*.
- We don't need to check the name to set the value, because that's already done inside the `CardName` enumerated type.
- Finally, we don't need to separately store the value, because we can just call the value() method of the `CardName` enumerated type.

This is much simpler and avoids many possible errors – or, more strictly, *the errors will be spotted by the compiler and not reach the stage where code which is run will be executed.*

Using Card – the CardDeck

In practice, cards come in packs, or decks. Just like the `Bank` and `BankAccount` examples, we can use `Card` to build a `Deck` class. `Deck` will use `Card` internally. Here's our first version:

```
public class Deck {
     //moved this from earlier Card classes
     private static final int DECK_SIZE = 52;

     private Card[] deck = new Card[DECK_SIZE];
     private Random rnd = new Random();

     //Constructor determines how 'shuffled' our deck is
     public Deck(int shuffleNumber) {
          int deckCounter = 0;

          for (Suite s : Suite.values()) {
               for (CardName c : CardName.values()) {
                    deck[deckCounter] = new Card(s,c);
                    deckCounter ++;
               }
          }

          // Now shuffle the deck
          shuffleDeck(shuffleNumber);
     }


     // Our toString() method outputs the entire deck,
     // with one card per line
     public String toString() {
          String result = new String();

          for (Card c : deck) {
               result += c +"\n";
          }
          return result;
     }

     //Shuffle the deck by swapping cards shuffles
     //number of times.
     public void shuffleDeck(int shuffles) {

          for (int i = 0; i < shuffles; i++){
               int swapFrom = rnd.nextInt(DECK_SIZE);
               int swapTo = rnd.nextInt(DECK_SIZE);
               Card temp = deck[swapTo];
               deck[swapTo] = deck[swapFrom];
               deck[swapFrom] = temp;
          }
     }
```

```
    //Now create an unshuffled Deck, then shuffle it
    public static void main(String[] args){
         //Create a deck shuffled zero times
         Deck deck = new Deck(0);
         System.out.println(deck.toString());
         //Now shuffle 100 times
         deck.shuffleDeck(100);
         System.out.println();
         System.out.println(deck);
    }

}
```

We'll look at this in stages – first the constructor:

```
    public Deck(int shuffleNumber) {
         int deckCounter = 0;

         for (Suite s : Suite.values()) {
              for (CardName c : CardName.values()) {
                   deck[deckCounter] = new Card(s,c);
                   deckCounter ++;
              }
         }

         // Now shuffle the deck
         shuffleDeck(shuffleNumber);
    }
```

We're filling an array with cards and we use two nested loops – one loops over the suite values and the other (inner) one over the card values so we create 52 cards and fill the entire array. Notice that we defined a `values()` method for the `CardName` enumerated type, but didn't for `Suite` – and yet we are using one anyway! It turns out that you get a default one 'for free' – if you don't say otherwise, the elements of your enumerated type are labeled from 0 to *n*-1, where *n* is the number of elements in the type.
This generates a deck where all the cards are in order, which is not good – so we have another method `shuffleDeck()` (which we call in the constructor above):

```
    public void shuffleDeck(int shuffles) {

         for (int i = 0; i < shuffles; i++){
              int swapFrom = rnd.nextInt(DECK_SIZE);
              int swapTo = rnd.nextInt(DECK_SIZE);
              Card temp = deck[swapTo];
              deck[swapTo] = deck[swapFrom];
              deck[swapFrom] = temp;
         }
    }
```

This method randomly chooses two cards (the values `swapFrom` and `swapTo`) and just switches them over – it does this a number of times defined by the parameter shuffles.

The main program just creates a deck shuffled 0 times, prints it, then shuffles it 100 times, and prints it again.

```
          Deck deck = new Deck(0);
          System.out.println(deck.toString());
          //Now shuffle 100 times
          deck.shuffleDeck(100);
          System.out.println();
          System.out.println(deck);
```

We are actually missing some methods we would really need – for example to get cards from the deck, and replace them in the deck. Here’s a version with more methods and using and `ArrayList` to store the cards:

```
public class Deck2 {
     private static final int DECK_SIZE = 52;

     private ArrayList<Card> deck =
           new ArrayList<Card>();

     public Deck2() {

          // As before but this time add to the arraylist
          // We can call the Card constructor directly
          // as a parameter
          for (Suite s : Suite.values()) {
               for (CardName c : CardName.values()) {
                    deck.add(new Card(s,c));
               }
          }

          // Now shuffle the deck
          shuffleDeck();
     }
```

```
// Our toString() method outputs the entire deck,
// with one card per line - EXACTLY the same
// code as last time
public String toString() {
     String result = new String();

     for (Card c : deck) {
          result += c +"\n";
     }
     return result;
}

//Shuffle the deck by calling a pre-existing
//method - we don't have to write our own
public void shuffleDeck() {
     Collections.shuffle(deck);
}

//Return the size of a *full* deck
public static int getDeckSize() {
     return DECK_SIZE;
}

//Return the value of a random card in the deck –
//don't remove the card
public Card cutDeck() {
     return deck.get(new
          Random().nextInt(deck.size()));
}

//Return - and remove - the top card
public Card dealCard() {
     Card nextCard = deck.get(0);
     deck.remove(0);
     return nextCard;
}

//Return a card to the back of the deck
public void returnCard(Card rtn){
     deck.add(rtn);
}

//How many cards are in the deck now?
public int getNoCardsInDeck() {
     return deck.size();
}
```

```
public static void main(String[] args){
          //Create a new shuffled deck and do some tests
          Deck2 deck = new Deck2();
          System.out.println(deck);

          System.out.println();

          System.out.println("Cut: "+ deck.cutDeck());

          Card dealt = deck.dealCard();
          System.out.println("Top card is: " + dealt);
          System.out.println("There are " +
                    deck.getNoCardsInDeck() +
                    " in the deck at the moment");
          deck.returnCard(dealt);
          System.out.println(deck);
     }
}
```

This adds methods to remove and add cards back into the deck (`dealCard()`, `returnCard()` and `cutDeck()`), as well as return information about the deck (e.g. `getDeckSize()` and `getNoCardsInDeck()`). Also, unlike the previous version, we don't have to write out own shuffle algorithm: there is *already* one which works on `ArrayLists` (called `Collections.shuffle()`)

Comparing Cards

One thing we cannot do yet but which is key to most uses of cards is to be able to compare and sort them. This is obviously important but it's going to take a few tricky steps to manage.

How Do We Compare Cards?

This is the first decision we need to take – it's pretty obvious that a Ten is higher than a Three – but is a Three of Clubs higher than a Three of Hearts? We haven't considered the order of the suites at all. (There's the other problem that different games use different values for cards – but we're going to assume an Ace is the lowest and all picture cards - Jack, Queen, King - and Ten all have the same value.)
To work out the sorting of the suites we first note that – traditionally – the order (from high to low) is: Spades, Hearts, Diamonds, Clubs. We've already said that we get a 'free' `values()` method when we defined the Suite enumerated type. So we could just make sure we define the suites in the right order (lowest to highest) and we'll get an ordering automatically. But we don't like leaving things like this implicit (i.e. not expressly written down) so we prefer to spell it out:

```
public enum SuiteCompare {
     SPADES(4), HEARTS(3), DIAMONDS(2), CLUBS(1);

     private final int value;

     SuiteCompare(int suiteValue){
          value = suiteValue;
     }

     public int value() {
          return value;
     }

}
```

This now explicitly spells out what the relative numerical values of the suites are – we now just need to think of a way of using it.

The compareTo() Method

While it would be great to come up with some way of just using the > and < operators (as well as ==, !=, <=, >=) to compare cards, we can't: the best we can do is come up with a method that works in a similar way to the `equals()` method we use for strings. We are going to write a method `compareTo()` that returns 0, 1, or -1 depending if two cards are equal, less than or greater than each other. This may seem an odd choice but bear with me – it will make sense. To do this, we're going to define a new class called `CardCompare` that has our new method and uses our new enumerated type for suites – here's the code:

```
public class CardCompare implements
                    Comparable<CardCompare> {

     private final SuiteCompare suite;
     private final CardName name;

     public CardCompare(SuiteCompare suiteVal,
               CardName cardVal){
          name = cardVal;
          suite = suiteVal;
     }

     public SuiteCompare getSuite() {
          return suite;
     }

     public CardName getName() {
          return name;
     }
public int getValue() {
          return name.value();
     }

     public int getSuiteValue() {
          return suite.value();
     }

     public String toString() {
          return (name + " of " + suite);
     }

     public int compareTo(CardCompare card) {

          if(card.getValue() < name.value()) {
               return 1;
          } else if (card.getValue() > name.value()) {
               return -1;
          } else if (card.getSuiteValue() <
                     suite.value()) {
               return 1;
          } else if (card.getSuiteValue() >
                     suite.value()) {
               return -1;
          } else {
               return 0;
          }
     }
}
```

The `compareTo()` method first checks the card values; and if one is greater than the other uses that. Next, if the values are equal, it uses the suites to compare values. Only if the cards are equal (both value and suite) does it return zero

We can use it like this:

```
CardCompare kingSpades =
     new CardCompare(SuiteCompare.SPADES, CardName.KING);

CardCompare eightClubs =
     new CardCompare(SuiteCompare.CLUBS, CardName.EIGHT);


if (eightClubs.compareTo(kingSpades) == -1) {
     //Should be true because kingSpades higher than
eightClubs
```

A little clumsy but better than manually calling `getValue()` and `getSuite()` for each card and checking each part in turn every time you compare two cards.

Sorting

The reason why we specifically wrote a method called `compareTo()` which does what it does is at the very top of the class:

```
public class CardCompare implements
Comparable<CardCompare>
```

We haven't seen this before. The keyword implements says that our class `CardCompare` contains implementations of the methods listed in a Java Interface – in this case the interface `Comparable`. I'm not going to explain how to write an interface (that's for a later module) – but it's basically just a list of the first lines of each method: the lines that say what the method name is, the arguments and the return type. Essentially `Comparable` in this case contains a line like this:

```
public int compareTo(CardCompare card);
```

So nothing about how the method works at all. By saying

```
public class CardCompare implements
Comparable<CardCompare>
```

we are saying that our class `CardCompare` contains an implementation of this interface (which of course it does). This is important because there's a method called `Collections.sort()`. This method will sort into order *any* class that implements the Comparable interface – it works because it knows that *any* class that does this *must* have the `compareTo()` method, which defines what 'in order' means for that particular class. In our case, `compareTo()` defines what it means for two cards to be 'in order'. So we can use this `Collections.sort()` method (which, by the way, is written to be very efficient – you will see what this means in a later module) to put our cards back into sorted order without having to actually write code to do it. This is one of the great strengths of Object Oriented programming – the ability to reuse code like this so generally.

Here's the new (and final) class:

```
public class Deck3 {
     private static final int DECK_SIZE = 52;

     private ArrayList<CardCompare> deck = new
ArrayList<CardCompare>();
public Deck3() {
          for (SuiteCompare s : SuiteCompare.values()) {
               for (CardName c : CardName.values()) {
                    deck.add(new CardCompare(s,c));
               }
          }
          shuffleDeck();
     }

     public String toString() {
          String result = new String();

          for (CardCompare c : deck) {
               result += c +"\n";
          }
          return result;
     }

     public void shuffleDeck() {
          Collections.shuffle(deck);
     }

     public static int getDeckSize() {
          return DECK_SIZE;
     }

     public CardCompare cutDeck() {
          return deck.get(ne
                Random().nextInt(deck.size()));
     }

     public CardCompare dealCard() {
          CardCompare nextCard = deck.get(0);
          deck.remove(0);
          return nextCard;
     }

     //NEW - Sort the deck into order
     public void order() {
          Collections.sort(deck);
     }

     public void returnCard(CardCompare rtn){
          deck.add(rtn);
     }

     public int getNoCardsInDeck() {
          return deck.size();
     }

}
```