**Arranging Objects in Arrays**

It is very frustrating to realize, after you are well into a project, that the way you are approaching a problem will not work. If you find yourself having to start over or change direction in the middle of writing a program, you will probably realize that the problem could have been foreseen if you had delayed coding until you had devised a plan. When you find yourself in this situation (and you will), the worst thing you can do is to try and force the program with a poor design. It will only take a few experiences like this to persuade you to plan first!

Efficiency should be your guide when instantiating objects; avoid writing redundant code. For example, if you instantiated just two objects for a class with two attributes and three methods, it would require 10 separate statements for both objects to access each attribute and method, as the following code segment illustrates. This brute force approach could quickly become a very tedious typing chore if the class contained additional variables or methods—and, if more objects were instantiated, the code would very quickly become cluttered.

anObject1.getAttribute1();  
anObject1.getAttribute2();  
anObject1.method1();  
anObject1.method2();  
anObject1.method3();  
anObject2.getAttribute1();  
anObject2.getAttribute2();  
anObject2.method1();  
anObject2.method2();  
anObject2.method3();

If you find yourself typing a series of statements with repeatable variations, remember the old adage, "if two or more, use a for."  Wouldn’t this be a good situation in which to apply an array?

Java offers you an alternative worth considering: an array of objects. The brute force approach shown above could be reduced to a for loop containing only five statements if using an array of objects. The same principle applies to using arrays instead of individual variables.

If the code segment shown above was extended from two to 10 objects, how many statements would be required?  Imagine maintaining such a program!  With an array of objects, instead of 50 statements, only five would be necessary. In fact, regardless of the number of objects instantiated (e.g., 2, 10, 50, 125, 5000, etc.), only five statements within a for loop would be required in this example. Accomplished programmers use elegant solutions like arrays of objects over brute force approaches, to save time and write efficient code.

Programmers quickly run into problems if design issues are not considered before the first line of code is written. Time is one thing you never have enough of, so you should avoid needlessly wasting it with poor programming habits. If you do not have time to redo something, make a plan before you begin every project. Ten minutes spent away from the computer, developing a pseudocode algorithm and a class diagram, can save you hours of frustration.

The following class definition is designed to represent a playing card:

public class Card   
{   
  private String mySuit;   
  private int myValue;   
  private String[] cardNames =    
      {   
        "Deuce", "Three", "Four", "Five", "Six", "Seven",   
        "Eight", "Nine", "Ten", "Jack", "Queen", "King", "Ace"   
      };   
  
  /\*\*   
   \* Creates a standard playing card.   
   \*   
   \* @param suit   A String, either "spades", "hearts", "diamonds", or "clubs"   
   \* @param value  An int from 2 through 14   
   \*/   
  public Card( String suit, int value )   
  {   
    mySuit = suit;   
    myValue = value;   
  }   
  
  /\*\*   
   \* Returns a String representation of this card   
   \*   
   \* @return  A String of the form "<card name> of <suit name>"   
   \*/   
  public String name()   
  {   
    return cardNames[ myValue - 2 ] + " of " + mySuit;   
  }   
}

Playing cards normally come in packs of 52 (not counting the "jokers") arranged in four *suits* (spades, hearts, diamonds, and clubs), each containing thirteen cards whose names are stored in the cardNames array in the above class definition and whose *pip values* range from 2 through 14.

Having processed the class definition for Card, the Java compiler creates the Card data type. Of course, as soon as the system knows about a new data type, we may create arrays of objects of that data type. Just as a one-dimensional array of ints has data type int[], so a one-dimensional array of Cards has data type Card[]. We may therefore create an array of Cards and initialize it (using the keyword new) like this:

  Card[] cardArray;   
  cardArray = new Card[ 52 ];

or in a single, combined statement, like this:

  Card[] cardArray = new Card[ 52 ];

Recall from our [earlier work](https://www.eimacs.com/eimacs/mainpage?epid=E1971625930&cid=162149#ArrayInit) with arrays that at this stage the array cardArray has been initialized, but its elements have not. Under such circumstances, each element of this array of Cards contains null (just as was the case for [uninitialized elements](https://www.eimacs.com/eimacs/mainpage?epid=E2286991542&cid=162149#ArrayElUninit) of arrays of Strings). If we attempt to access any property of an element of this array before that element has been initialized, a *NullPointerException* error will be generated.

public static void main( String[] args )

{

Card[] hand = new Card[ 5 ];

hand[ 0 ] = new Card( "clubs", 4 );

hand[ 1 ] = new Card( "clubs", 7 );

// these statements execute without error

System.out.println( hand[ 0 ].name() );

System.out.println( hand[ 1 ].name() );

// this one causes an exception

System.out.println( hand[ 2 ].name() );

}

Alternatively, we can use the shorthand technique we introduced earlier for initializing both an array and its elements at the same time. Here, for example, we create and initialize an array of five Cards and at the same time initialize the individual Card object elements too:

    public static void main( String[] args )   
    {   
      Card[] pokerHand =    
        {   
          new Card( "clubs", 4 ),   
          new Card( "clubs", 7 ),   
          new Card( "spades", 12 ),   
          new Card( "hearts", 7 ),   
          new Card( "diamonds", 2 )   
        };   
  
      int i;   
      for ( Card card : pokerHand )   
        System.out.println( card.name() );    
    }

Many program design considerations boil down to how to pass parameters to objects. Do you instantiate an object through the constructor's private instance variables, do you pass values directly to a method's parameter list, or do you use some combination? There are often different solutions to achieve the same goal; however, each choice has its own set of consequences for method design. Fortunately, these choices become less complicated because there are basically only four kinds of methods to worry about: [accessors](javascript:void(0);) (methods that accesses private instance variables, but do not change them), [mutators](javascript:void(0);) (methods that change the value of private instance variables), [getters](javascript:void(0);) (methods which return a value to an invoking statement), and [setters](javascript:void(0);) (methods which only assign values to variables or objects).