The instance variables of the Person class — like those of *every* class in the Advanced Placement Computer Science Java subset — are declared to be private. Consequently, they can only be accessed directly from within the Person class constructor and its instance methods. We need a way, however, for the outside world to gain access to the values of these variables. To that end, we introduce additional instance methods, each of which has the simple task of returning the value of one of the instance variables. The following code, for example, includes a method whose sole job is to return the value of the instance variable myHair:

  public class Person   
  {   
    private double myHeight;   
    private double myWeight;   
    private String myHair;   
  
    public Person( double height, double weight, String hair )    
    {   
      myHeight = height;   
      myWeight = weight;   
      myHair = hair;   
    }   
  
    public void sleep( int hours )   
    {   
      for ( int i = 0 ; i < hours ; i++ )   
        System.out.println( "Sleeping ... " );   
    }   
  
    public void talk()   
    {   
      myWeight -= 0.01;   
      System.out.println( "Talking ... " );   
    }   
  
    public String getHair()   
    {     
      return myHair;   
    }   
  }

public static void main( String[] args )

{

  Person eric = new Person( 1.57, 67.2, "brown" );

  Person jenny = new Person( 1.55, 56.8, "blond" );

  System.out.println( "Eric's hair is " + eric.getHair() );

  System.out.println( "Jenny's hair is " + jenny.getHair() );

}

 Eric's hair is brown   
Jenny's hair is blond

(Notice that this code uses the abbreviated form of initializing the variables eric and jenny.) Instance methods such as getHair are commonly referred to as accessor methods, since their purpose is to allow access to the values of the instance variables. Note that, just as we have adopted the practice (mentioned [earlier](http://pages.eimacs.com/eimacs/mainpage?epid=E1942114632&cid=162149#MyFields)) of using instance variables that start with my..., so we adopt the convention of giving the corresponding accessor methods names that start with get.... Thus, unless there is some pressing reason to act differently, we will use getVar as the name of the accessor method for an instance variable myVar.

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.

**Exercise 104**

What is the value of c after the following code is executed?

  Card t = new Card( "diamonds", 7 );   
  String c = t.name();

### Exercise 104

The value of c is the String "Seven of diamonds".

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](http://pages.eimacs.com/eimacs/mainpage?epid=E2306666313&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](http://pages.eimacs.com/eimacs/mainpage?epid=E2153902256&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. (To witness this happening, click the **Run** button beneath the following code.)

  public class Card   
  {   
    private String mySuit;   
    private int myValue;   
  
    public Card( String suit, int value )   
    {   
      mySuit = suit;   
      myValue = value;   
    }   
  
    public String name()   
    {   
      String[] cardNames =    
        {   
          "Deuce", "Three", "Four", "Five",   
          "Six", "Seven", "Eight", "Nine", "Ten",   
          "Jack", "Queen", "King", "Ace"   
        };   
  
      return cardNames[ myValue - 2 ] + " of " + mySuit;   
    }   
  }   
  
  public class MainClass   
  {   
    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() );   
    }    
  }

Four of clubs   
Seven of clubs

Exception in thread "main" java.lang.NullPointerException   
  
at MainClass.main(MainClass.java:17)

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 class Card   
  {   
    private String mySuit;   
    private int myValue;   
  
    public Card( String suit, int value )   
    {   
      mySuit = suit;   
      myValue = value;   
    }   
  
    public String name()   
    {   
      String[] cardNames =    
        {   
          "Deuce", "Three", "Four", "Five",   
          "Six", "Seven", "Eight", "Nine", "Ten",   
          "Jack", "Queen", "King", "Ace"   
        };   
  
      return cardNames[ myValue - 2 ] + " of " + mySuit;   
    }   
  }   
  
  public class MainClass   
  {   
    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() );    
    }   
  }