**Static Variables and Constants**

Code breaking machines and their human handlers played a significant role in World War II. Nazi Germany used the Enigma cipher machine to encrypt and decrypt military messages.

Hundreds of mathematicians, crossword puzzle experts, chess champions, and cryptographers were stationed on a rambling country estate in Bletchley, England, home of the British code-breaking efforts during WWII. Alan Turing, the most famous member of the team, is now recognized as one of the founders of computer science.

Codes of several Axis countries were deciphered, most famously the Enigma cipher machine. Colossus, one of the first digital electronic computers, was developed at Bletchley Park and used to break a number of enemy codes. Unfortunately, to maintain secrecy, Colossus and its blueprints were destroyed after the war. The general public only learned in the 1970s of its existence and the work of the unheralded code breakers.

### Part 1

Class variables have a much broader scope than either instance or local variables. Consequently, any change (accidental or on purpose) to a class variable will affect all instances of that class. This can be avoided by using constants. If non-static variables (i.e., instance or local variables) are declared as constants, you must pay close attention to their scope.

Writing in ALL CAPS in an email is regarded as shouting (and somewhat rude), although it certainly gets your attention. Similarly, capitalizing Java constants is a good convention to follow because it alerts you to their special nature. However, **capitalizing a constant is not required**, so do not let that throw you off if you see it on a test or in program. The keyword final is what is important, not the case of the identifier.

We have [seen](https://www.eimacs.com/eimacs/mainpage?epid=E2212929266&cid=162149#ClassStatic) that the presence of the static keyword distinguishes a class method from an instance method. The static keyword is also used to identify *class variables*. A class variable is available both to instance and to class methods and, as we demonstrate shortly, it may even be accessed outside the class in which it is declared.

Our first example illustrates the situation in which a class variable is referenced in a class method of the same class. In this case, the class variable today is initialized in the definition of the MainClass class and is referenced in the main method of the same class.

public class MainClass   
{   
  private static String today = "Monday";   
  
  public static void main( String[] args )   
  {   
    System.out.println( today );   
  }   
}

Our second example has a class variable, testN, that is referenced in an instance method of the same class (namely, the Test class). In this case, the instance method testMe increments the class variable each time the method is called.

public class Test   
{   
  private static int testN = 0;   
  
  public Test()   
  {   
  }   
  
  public void testMe()   
  {    
    System.out.println( testN );   
    testN++;   
  }   
}   
  
public class MainClass   
{   
  public static void main( String[] args )   
  {   
    Test t1 = new Test();   
    t1.testMe();   
    t1.testMe();   
  
    Test t2 = new Test();   
    t2.testMe();   
    t1.testMe();   
    t2.testMe();   
  }   
}

If you click the **Run** button, you will perhaps be surprised to discover that both instances, t1 and t2, are accessing the same class variable, testN. This is a distinguishing characteristic of class variables. There is just one copy of each class variable, and that one copy is shared by all the instances of the class. As you will recall, the situation is very different for instance variables. Each instance of a class has its own copy of each of the class's instance variables.

On the previous page we learned how to declare and initialize class variables, and we saw examples in which the value of a class variable was changed as code execution proceeded. Sometimes, however, it is desirable to prevent modification of a class variable. That is, there are sometimes occasions when we want to ensure that, once a class variable has been initialized, its value can never be changed. The way to do this is to include — along with the modifier keyword static (which makes it a class variable) — the modifier keyword final. An unchangeable class variable is referred to as a *class constant*. Here is an example:

public class Test

{

private static final int n = 5;

public Test()

{

}

public void getN()

{

System.out.println( n );

}

}

public class MainClass

{

public static void main( String[] args )

{

Test t1 = new Test();

t1.getN();

t1.getN();

Test t2 = new Test();

t2.getN();

t1.getN();

}

}

Many Java programmers follow the convention that the name of a class constant uses only capital letters, as in this code fragment:

public class MainClass   
{   
  private static final String TODAY = "Saturday";   
  
  public static void main( String[] args )   
  {   
    System.out.println( TODAY );   
  }   
}

The purpose of the convention is to make the variable stand out from the surrounding code. By being made to notice it in this way, you — and any other programmers who might touch your code in the future — will be more likely to remember that this is a variable whose value cannot be changed.

Class variables don't have to be private. By declaring a class variable using the modifier keyword public instead of private, we allow that variable to be accessed directly from outside the class in which it is declared. Such a reference is achieved by writing the class name, followed by a dot, followed by the name of the class variable, as in the following code:

public class Test

{

public static final int N = 5;

public Test()

{

}

}

public class MainClass

{

public static void main( String[] args )

{

Test t1 = new Test();

System.out.println( Test.N );

}

}

The final modifier keyword may also be used when declaring variables that are not class variables, that is, variables whose declarations do not include the static modifier. The effect of including the final modifier is to make it impossible to change the value of the variable, once it has been initialized. Furthermore, the absence of the static modifier has a profound effect upon the accessibility of the variable; it means that the variable is subject to the rules we listed [earlier](https://www.eimacs.com/eimacs/mainpage?epid=E2163739961&cid=162149#RulesVarDec), before we had told you about class variables. Each time the innermost block of code that contains the variable declaration is executed a new instance of the variable is created, and its value is only accessible from the point where it is initialized to the end of that block of code.

Consider the following code, for example:

public class Family   
{   
  ArrayList<FamilyMember> myMembers;   
  String    myFamilyName;   
  
  public Family( String lastName )   
  {   
    myFamilyName = lastName;   
    myMembers = new ArrayList<FamilyMember>();   
  }   
  
  public void add( FamilyMember famMem )   
  {   
    myMembers.add( famMem );   
  }   
  
  public String toString()   
  {   
    final int FAMILY\_SIZE = myMembers.size();   
  
    String s = "The " + myFamilyName + " family:\n";   
  
    for ( int i = 0 ; i < FAMILY\_SIZE ; i++ )   
      s += myMembers.get( i );   
  
    return s;   
  }   
}   
  
public class FamilyMember   
{   
  final String[] FIELD\_NAMES = { "First name", "Date of birth" };   
  String[] myData = new String[ FIELD\_NAMES.length ];   
  
  public FamilyMember( String firstName, String dob )   
  {   
    myData[ 0 ] = firstName;   
    myData[ 1 ] = dob;   
  }   
  
  public String toString()   
  {   
    final int NUM\_OF\_FIELDS = FIELD\_NAMES.length;   
  
    String s = "";   
  
    for ( int i = 0 ; i < NUM\_OF\_FIELDS ; i++ )   
    {   
      s += "  " + FIELD\_NAMES[ i ]  + ": ";   
      s += myData[ i ] + "\n";   
    }   
  
    return s + "\n";   
  }   
}

public class MainClass

{

public static void main( String[] args )

{

Family f = new Family( "Smart" );

f.add( new FamilyMember( "Farley", "10/31/1963" ) );

f.add( new FamilyMember( "MerryAnn", "7/1/1965" ) );

f.add( new FamilyMember( "Luke", "2/29/1988" ) );

System.out.println( f );

}

}

In this code, the variable FAMILY\_SIZE is accessible only in the body of the toString method in the Family class, the variable FIELD\_NAMES is accessible throughout the definition of the FamilyMember class, and the variable NUM\_OF\_FIELDS is accessible only in the body of the toString method in the FamilyMember class. Notice that, just as we did in the case of class constants, we are using capital-letters-only names for these final block variables. Notice also that the initialization of FIELD\_NAMES is "hard-coded", whereas the values to which FAMILY\_SIZE and NUM\_OF\_FIELDS are initialized are calculated dynamically.

There are various reasons for using such variables. These include making it less troublesome for you (or another programmer) to update the code in light of changing circumstances. Suppose, for example, you decided to add another field to the FamilyMember class (a middle name field, for example). Then you would have to make a couple of small changes to the class constructor and change the (hard-coded) initialization of the FIELD\_NAMES variable. But all the rest of the code could remain untouched. Furthermore, if sensible and informative variable names are chosen, the use of final block variables can make the structure of your code much easier to grasp; in a way, the variable names provide a level of self-documentation within your code.

This short section deals with the concepts of *mutability* and *aliasing*. Although these are not explicitly mentioned as belonging to the Advanced Placement Java subset, they can be a source of great confusion to any programmer who is not aware of their existence.

If an object is such that the values of its instance variables can be changed after it is created, then the object is said to be *mutable*. If an object's instance variable values are unchangeable after it is created, then that object is said to be *immutable*.

For example, objects of the following PersonA class will be immutable:

public class PersonA   
{   
  private String myName;   
  
  public PersonA( String name )   
  {   
    myName = name;   
  }   
  
  public String getName()   
  {   
    return myName;   
  }   
}

This is because the only thing that sets the value of the instance variable myName is the class constructor. Furthermore, although the class has an accessor method that returns the value of the instance variable myName, that value is a String and Strings are constants — they cannot be changed.

On the other hand, objects of the following PersonB class will be mutable because the instance method setName makes it possible for us to change the value of the instance variable myName after a PersonB object is created:

public class PersonB   
{   
  private String myName;   
  
  public PersonB( String name )   
  {   
    myName = name;   
  }   
  
  public String getName()   
  {   
    return myName;   
  }   
  
  public void setName( String newName )   
  {   
    myName = newName;   
  }   
}

It can be quite challenging to determine whether or not the values of a class's instance variables can be changed. There are, however, a number of rules of thumb:

* If one or more instance variables have corresponding modifier set... instance methods, then objects of the class are mutable.
* If each instance variable has a primitive data type or is a String and the class has no modifier instance methods, then objects of the class are immutable.
* If there is an instance variable that is an instance of Object or one of its subclasses, if there is a corresponding accessor get... instance method, and if the object that is the value of the instance variable is mutable, then objects of the class are mutable. (For example, any class with an instance variable whose value is an ArrayList and that has a corresponding accessor method will have mutable objects, because the ArrayList can be changed by "getting" it using the accessor method and then modifying it using its add method.)

Clearly, our programs can be much more dynamic if the state of the objects we create can be changed while execution of the program proceeds. Significant dangers arise, however, if mutable objects are in play when *aliasing* is also involved. Aliasing occurs when two or more different variables store one and the same object.

But first, consider this code fragment, which involves a (mutable) PersonB object. Study the code and try to figure out what the output will be. Then run the program to see how accurate your prediction was.

public class MainClass   
{   
  public static void main( String[] args )   
  {   
    PersonB a, b;   
  
    a = new PersonB( "fred" );   
    b = new PersonB( "fred" );   
  
    System.out.println( a.getName() );   
  
    b.setName( "eric" );   
  
    System.out.println( a.getName() );   
  }   
}

Here is almost exactly the same code again. The only difference is that this time, instead of creating two PersonB objects each having the name "fred", we simply assign to the second variable, b, the PersonB object we have just created and stored in the first variable, a. Once again we change b's name. Will the output be different this time? What do you think? Run the program to test your prediction.

public class MainClass   
{   
  public static void main( String[] args )   
  {   
    PersonB a, b;   
  
    a = new PersonB( "fred" );   
    b = a;   
  
    System.out.println( a.getName() );   
  
    b.setName( "eric" );   
  
    System.out.println( a.getName() );   
  }   
}

You should find that as a result of changing b's name, a's name has been changed too. Of course, since a and b are actually the same object, you should have predicted that this would happen. The variables a and b are said to be aliases, since they store exactly the same object — they are two different labels for one and the same thing.

Aliasing can be very useful in certain circumstances, but when one or more of the objects involved is mutable, unexpected behavior can occur if we are not very careful.

Writing in ALL CAPS in an email is regarded as shouting (and somewhat rude), although it certainly gets your attention. Similarly, capitalizing Java constants is a good convention to follow because it alerts you to their special nature. However, **capitalizing a constant is not required**, so do not let that throw you off if you see it on a test or in program. The keyword final is what is important, not the case of the identifier.

### Part 2

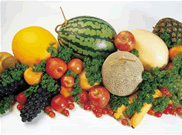
Class variables provide some important new programming capabilities. Several scenarios are presented below to help clarify the big picture.



**Scenario 1:** Consider the information you have to fill in on the answer sheet for a standardized test like an AP Exam. The answer sheet object provides fields (which are analogous to private instance variables) for you to store your unique information such as name, ID number, grade level, gender, etc. These data belong to you and nobody else, but other information that is requested is shared in common with everyone else. Everyone bubbles in the same exam name, school code, etc. (these shared data are analogous to Java’s class variables). Everyone uses the same data, and class variables increase efficiency.



**Scenario 2:** When you go to apply for a driver’s license, there are often a lot of people (objects) sitting around waiting their turn, rather than standing in line. When the receptionist asks “Who’s next?,” how does everyone know when to step up to the desk? If each person (individual object) assigned themselves a number when they entered the office, total chaos would result. Instead, there is often a little machine with a notice that says “Take a Number,” which dispenses tickets. Because the tickets are numbered sequentially and provided to everyone waiting (the class of objects), everyone is helped in their correct turn. The simple ticket dispenser functions exactly like a class variable. Each person (instance of an object) utilizes the same ticket dispenser, thereby maintaining an orderly process in the office.

© 2007 Jupiterimages Corporation

**Scenario 3:** Consider the inventory of a grocery store chain. Each store (object) is linked by a network, and the prices at every store are always the same. Individual store managers have the ability to adjust prices depending on product availability and demand. If the manager of Store A decides to increase the price of seedless watermelons from $3.89 to $3.99, all it takes is changing the price variable for watermelons at his store and the price is automatically adjusted at all of the other stores. Or the manager of Store B decides that cantalopes are priced too high at $1.59 and drops the price to $1.49. The price for cantalopes is instantly changed for the product in every store.