A Step-by-Step User’s Guide to Implementing the Group Class in Java

Victor D. Wiley
The University of Texas at Austin
September 19, 2000

Abstract
This is a user guide intended to be a step-by-step instruction of how to create a group using the Java based Group class definition as implemented by Victor Wiley. This guide will demonstrate each step of the process using the Symmetric Group on $n$ letters ($S_n$) as an illustration. The methods defined for $S_n$ during the step-by-step instructions are meant to be the minimal set of necessary methods and are not intended to restrict the user from creating other useful methods. For additional methods that have successfully been applied to $S_n$, refer to the SymmetricGroup User’s manual.

Contents

1 Know the Properties of a Group 2
2 Identify the Group 2
  2.1 Determine the element representation .......................... 2
  2.2 Define the group operation ........................................ 3
3 Code the group 3
  3.1 Opening declaration ............................................. 3
  3.2 Data members ................................................... 4
  3.3 Constructors .................................................... 4
4 Group methods 5
  4.1 getIdentity .................................................... 5
  4.2 operate ......................................................... 6
  4.3 invert ........................................................... 6
  4.4 equals .......................................................... 7
1 Know the Properties of a Group

This chapter provides the basic definition of a group as used in the Group Class. This serves mainly as a review for users who are not familiar with the properties of a group or who may have forgotten.

Definition 1 A group is a set of elements, G, that satisfy the following:

- G contains an identity element,
- G contains the inverse of each of its elements, and
- G is closed under an associative law of composition (a group operation).

These properties serve as the baseline methods that must be defined in order to successfully extend the Group Class.

2 Identify the Group

This sounds a little silly, but the next step makes a lot of sense. Clearly, it will be near impossible to create a group based on the Group Class if you have no idea what representation your elements will take or what operation will be applied to these elements. If the group being implemented is a well-known group (e.g. the integers under addition, the invertible matrices under matrix multiplication, the symmetric group under function composition) then this step should be fairly “easy” to complete. However, if the group being implemented has not already been defined, then a little time spent deciding the appropriate representation is worth the investment. Once created, the group can be tested using the methods provided within the Group Class.

2.1 Determine the element representation

The specific form used to represent an element in a group plays an important role. The form should be one that can be easily manipulated within the Java Object framework. Specifically, the form should be conducive to the associative law of composition defined for the group. As an example, the symmetric group has two possible forms. Examples 2 and 3 show the two forms. The long form
can be represented as an array of integers. The indices of the array represent specific letters and the values of the array represent the image of the letter (the top and bottom rows, respectively, in Example 2). The cyclic form contains a dynamic array of arrays of integers. Each array of integers contains a letter with the successor being its image. Letters whose images are themselves are dropped from the notation as in Example 3. Both of these forms contain arrays of integers which are easily manipulated, but the cyclic form holds these arrays in a dynamic structure that makes retrieval of the data a bit more cumbersome. However, each form has advantages when performing operations on $S_n$.

*Long Form*

Example 2  
\[
\begin{pmatrix}
1 & 2 & 3 & 4 & 5 & 6 \\
3 & 6 & 5 & 4 & 1 & 2
\end{pmatrix}
\]

*Cyclic Form*

Example 3  
\[(1, 3, 5) (2, 6) (4) = (1, 3, 5) (2, 6)\]

### 2.2 Define the group operation

The specific group operation performed influences the data representation. Obviously, the operation needs to be applicable to the work that will be performed by the group, but the coding of the operation can directly influence the performance of a program. For the Symmetric Group, if most of the work involves “multiplication” of two elements, then the long form manipulation is straightforward. On the other hand, if conjugation is the primary workhorse then the cyclic form manipulation is easiest.

### 3 Code the group

Now that the form of the elements and the group operation has been decided, the group needs to be coded into Java. For Java applications, a hierarchical structure for objects exists. All objects are derived from the base class `java.lang.Object`. Similarly, the group that is being defined will be a derivation using the `Group Class`. The syntax for this derivation would look like Example 4 where the bold faced lettering denotes reserved words in Java.

#### 3.1 Opening declaration

The opening declaration simply defines the accessibility (public, protected, or private), the type (class, int, void, and so on), and any extra descriptions (extends, implements). In Example 4 the `public` statement makes the class, named `SymmetricGroup`, available for other classes to reference. The `class` statement describes how to treat the code and the `extends` statement identifies the derivation class, `Group` in this case.
Example 4  public class SymmetricGroup extends Group

After declaring the attributes of the class, the data members that make up an element of the group need to be defined.

3.2  Data members

The data members of an element store information about that specific element. The data members should contain all the relevant information about the element. Initially, having too many data members and overdescribing the element may be better than not enough and underdescribing the element. Once the rest of the methods have been coded, unnecessary data members can be removed. Data members’ accessibility can be declared as public, protected, or private, although they are usually private so that other classes can not alter them. Example 5 lists the individual data members associated with \( S_n \).

Example 5

```java
private int[] longForm;
private Vector cyclicForm;
private int numLetters;
private boolean cyclicFormValid;
```

As shown in Example 2, the long form of a Symmetric Group element can be represented by an array of integers. Since an element of \( S_n \) can also be represented in cyclic form, it is contained within the Vector object. However, the Vector object itself is not enough to describe the element in that the identity of \( S_n \), () “looks” the same as the identity of \( S_{100} \), (). Therefore, the cardinality of the symmetric group represented is included in the description of the element. Finally, since there are two forms possible, a data member that serves as an indicator for which form is being used is included.

3.3  Constructors

Once the data members of the group element have been defined, Java needs instructions on how to initialize the data members when an element is instantiated. Typically, an empty constructor is used to initialize objects to null or zero by default. Other types of constructors initialize parts of or all of the object based on data passed through the parameters list. Example 6 shows an empty constructor and Example 7 shows a constructor using an integer array. The array constructor makes use of the method setLongForm that has been defined as a support method within the SymmetricGroup class.

Example 6

```java
public SymmetricGroup() {
    longForm = new int[0];
    cyclicForm = new Vector();
    numLetters = 0;
    cyclicFormValid = false;
}
```
public SymmetricGroup( int[] initialTour ) {
    numLetters = initialTour.length;
    longForm = new int[ numLetters ];
    System.arraycopy( initialTour, 0, longForm, 0, numLetters );
    setLongForm( initialTour );}

Another useful example of a constructor is the copy constructor. A copy constructor takes an element of the group and can create a “light” or “heavy” copy of the object. A “light” copy of an element simply creates references to the original object’s data. A “heavy” copy will create a new physical memory copy of the original object. This is an important concept to keep in mind. If you make a “light” copy, any changes you make to the data of the original or the copy will show up in both objects. Example 8 shows the copy constructor for the Symmetric Group. Note that this is a “heavy” construction.

public SymmetricGroup( SymmetricGroup oldCopy ) {
    numLetters = oldCopy.numLetters;
    if ( oldCopy.cyclicFormValid ) {
        cyclicForm = new Vector( oldCopy.cyclicForm.size() );
        setCyclicForm( oldCopy.cyclicForm, oldCopy.numLetters );
    } else {
        longForm = new int[ oldCopy.longForm.length ];
        setLongForm( oldCopy.longForm );
    }
}

4 Group methods

Creating a derivation of the Group Class through the extends key word enables the user to inherit the methods and data members of the Group Class. In so doing, before being able to instantiate any objects with the constructors mentioned in Section 3.3 a number of methods must be defined. Specifically,

- getIdentity,
- operate, and
- invert.

4.1 getIdentity

The getIdentity method returns a new object that represents the identity element of the group. This method must be declared using the following syntax: public Group getIdentity(). This syntax coincides with the getIdentity declaration within the Group Class. The Symmetric Group version of this is shown in Example 9. Depending on the form used in the original element that calls this method, the identity element will be returned as a long or cyclic representation. An important note on this method is that it creates a Group object that will be treated as a Group object unless it is cast back into a SymmetricGroup.
object. The Group Class will understand the result of this method, but the SymmetricGroup class may not.

Example 9

```java
public Group getIdentity() {
    SymmetricGroup identityOfThis;
    if ( getCyclicFormValid() ) {
        identityOfThis = new SymmetricGroup( cyclicForm, numLetters );
        identityOfThis.cyclicForm.clear();
        return identityOfThis;
    } else {
        identityOfThis = new SymmetricGroup( longForm );
        for( int i = 0; i < identityOfThis.longForm.length; i++ )
            identityOfThis.longForm[ i ] = i;
        return identityOfThis;
    }
}
```

4.2 operate

The operate method represents the associative law of composition for the group. This method must be declared using the following syntax: `public Group operate( Group operand ).` This syntax coincides with the operate declaration within the Group Class. The method takes the group object calling the method and the Group object in the parameter list, performs the operation, and then returns a Group object. Once again, the object created is treated as a Group object. Example 10 shows the SymmetricGroup class implementation of this method. Note that this example shows that the method may be defined to simply be the result of another method called within the SymmetricGroup class. Reuse of code is one of the advantages of Object Oriented Programming.

Example 10

```java
public Group operate( Group operand ) {
    return getProduct( (SymmetricGroup) operand );
}
```

4.3 invert

The invert method takes the current element of the group and converts the element into its inverse. The method must be declared using the following syntax: `public void invert()`. One of the main differences in this method versus the previous two methods involves the object calling this method. Previously, the SymmetricGroup examples of the getIdentity and operate methods returned new Group objects while this method actually alters the data of the object itself. One note: the operate method could have altered the object calling the method by simply calling the appropriate SymmetricGroup method within the operate method. The SymmetricGroup class example of invert in Example 11 once again determines the form of the representation (long or cyclic) and then performs the appropriate actions and methods.
public void invert() {
    if ( cyclicFormValid ) invertCyclic();
    else if ( longForm.length != 0 ) invertLong();
    else System.out.println("Trying to invert something that is empty");
}

With these methods defined, the group can now be tested to see if a set of elements truly form a group. However, before the testing begins, one more method must be defined — the equals method.

4.4 equals

Every Object within java inherits an equals method. This default method compares the memory location of two objects. If the objects reference the same memory location than it returns true, otherwise it returns false. The method must be declared with the following syntax: public boolean equals( Object compare2 ). Once declared, simply provide the class with the means to declare two objects equivalent or not. This will effectively override the inherited Object equals method. In the SymmetricGroup class, this method compares the elements either letter-by-letter or cycle-by-cycle depending on the form, long or cyclic, respectively. Example 12 provides the SymmetricGroup class equals method. The first test is whether the Object being compared is a SymmetricGroup object. If it is a SymmetricGroup object, then it is recast as a SymmetricGroup object and the appropriate method is called.

    public boolean equals( Object compare2 ) {
        if ( !(compare2 instanceof SymmetricGroup ) ) return false;
        SymmetricGroup tempCompare = (SymmetricGroup) compare2;
        if ( tempCompare.getCyclicFormValid() )
            return cyclicEquals( tempCompare );
        return longEquals( tempCompare );
    }

5 Group Support Methods

There are a number of support methods that are inherited by any class that extends the Group Class. These support methods consist of tests for the various properties of a group defined in Definition 1 and can be called from within the derived group. The list of these includes the following:

- hasIdentity
- hasInverse
- isOperatorAssociative
- isOperatorClosed
- subGroupTest
• isOperatorCommutative
• abelianSubGroupTest

Each of these tests requires an array of objects that represent the subset to test. However, as a note of caution, the larger the subset passed into the method, the longer the test will take. This is especially so with the isOperatorAssociative and isOperatorClosed methods.

5.1 hasIdentity
The hasIdentity method takes a user-defined subset of objects and calls the getIdentity method of the derived group. It then compares the identity element with each element in the array until a match occurs. If a match occurs, the method returns true, otherwise false.

5.2 hasInverse
The hasInverse method takes a user-defined subset of objects, calls the invert method, and then tests to see if the subset contains the inverted object. If every object’s inverse is contained within the subset, then the method returns true, otherwise false.

5.3 isOperatorAssociative
The isOperatorAssociative method takes a user-defined subset of objects and calls the operate method twice using three elements (not necessarily distinct) of the subset at a time. If \((a \odot (b \odot c)) \neq ((a \odot b) \odot c)\) for any \(a, b, c \in \{\text{subset}\}\) then the method returns false, otherwise true. Clearly, as the subset grows large, this comparison will be time consuming.

5.4 isOperatorClosed
The isOperatorClosed method takes a user-defined subset of objects and calls the operate method using two elements (not necessarily distinct) of the subset at a time. If \((a \odot b) \not\in \{\text{subset}\}\) then the method returns false, otherwise true. Once again, as the subset grows large, this method can be time consuming.

5.5 groupTest
The groupTest method calls each of the four methods previously described. If one of the methods returns false, the groupTest returns false. Otherwise, the groupTest returns true.
5.6 subGroupTest

This method calls the hasInverse and isOperatorClosed methods. This method assumes that the user-defined subset of elements is taken from a set of elements known to form a group and makes use of the following Theorem:

**Theorem 13**  \( H \subseteq G \) is a subgroup of \( G \) if \( ab^{-1} \in H \forall a, b \in H \).

If one of the methods returns false, the subGroupTest returns false. Otherwise the subGroupTest returns true.

5.7 isOperatorCommutative

The isOperatorCommutative method takes a user-defined subset of objects and calls the operate method using two elements (not necessarily distinct) of the subset at a time. If \((a \otimes b) \neq (b \otimes a)\) then the method returns false, otherwise the method returns true. Use this method to test a known subgroup.

5.8 abelianSubGroupTest

The abelianSubGroupTest method takes a user-defined subset of objects and calls the subGroupTest method first. If the subset passes the subGroupTest, then the method calls the isOperatorCommutative method. If this method fails, the abelianSubGroupTest method returns false, otherwise true.