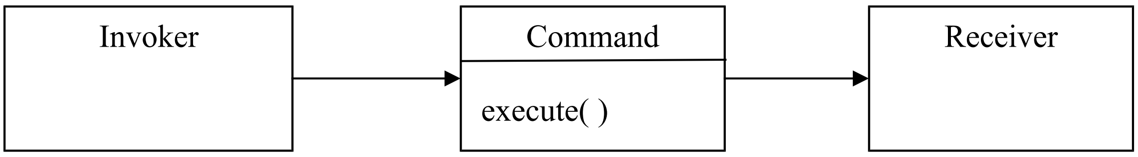
Command Pattern

The purpose of the Command Pattern is to encapsulate actions so that we can have a family of interchangeable actions, like "turn on", "turn off", or "blink". Benefits of this are:

1. The object that *invokes* a *command* isn't coupled to the object that *performs* the command (e.g., a LightBulb).



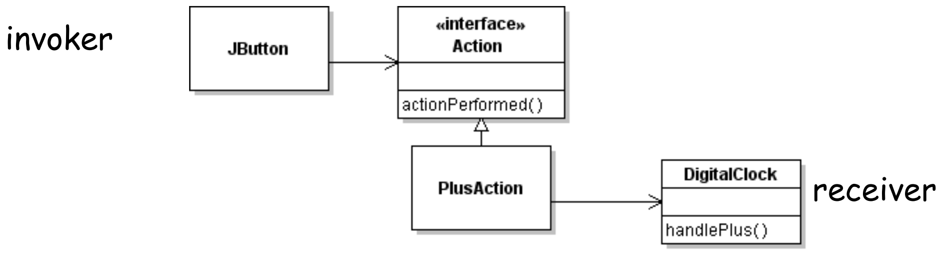
2. Commands all have the same interface. We can also *change* the Command without changing the code of the *Invoker*. Using *polymorphism*, the Invoker can invoke <u>any</u> Command we give it.

3. The commands can be reused, passed as parameters, etc.

Examples of the Command Pattern

1. ActionListener objects for Swing components.

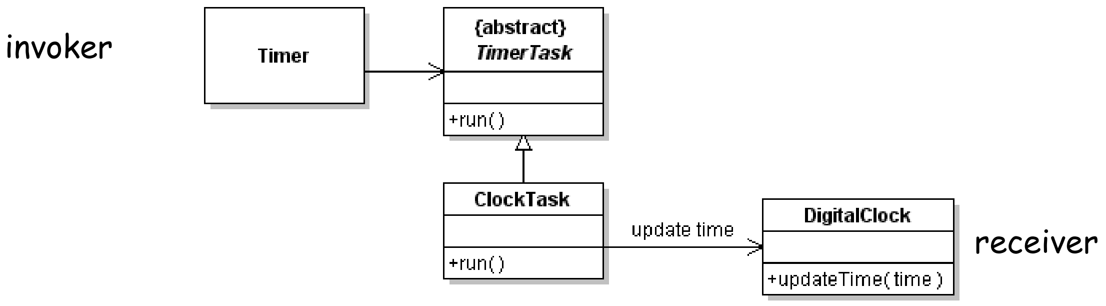
You have written classes that implement *ActionListener* to define actions to perform on another object (such as the Purse or DigitalClock).



2. Action objects. Actions can encapsulate more information than a plain *ActionListener*, such as a display name, an icon, and enable/disable status of a component. Usually you create Actions by extending *AbstractAction*.

The same Action object can be used for a Button, a MenuItem, or a TextField. The same action will be performed no matter which component invokes it.

3. Timer and TimerTask. A Timer *invokes* a TimerTask, that performs some action on another object. For example, updating the time in the Digital Clock.



Command Pattern for Calculator Operators

The calculator needs to remember the operator, like **+** or ***** and a way for the user interface to *set* the binary operator that calculator will perform. In a simple implementation, you may have used a char for this:

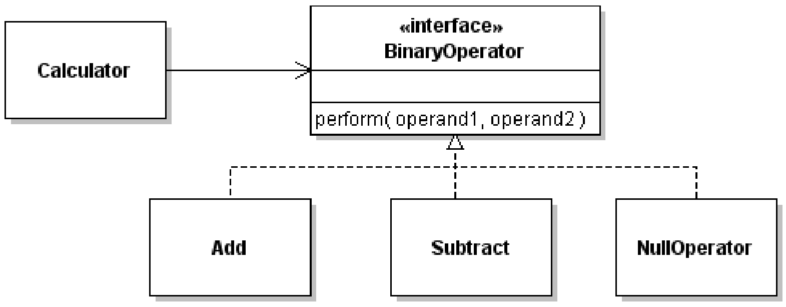
```
class Calculator {
   private double register1;
   private double register2;
   private char operator;

   public void handleOperator(char op) {
//TODO check calculator state and (maybe) perform current operation
      this.operator = op;
   }

   private void performOperation( ) {
      switch(operator) {
      case '+':
         register1 = register1 + register2;
         break;
      case '-':
         register1 = register1 - register2;
         break;
      ...
```

Instead of saving a **char** for the operation, use the *Command Pattern*.

1. Define a BinaryOperator interface to represent any binary operation, like + - * / or modulo (%).

2. When the user presses the key for a binary operator you set an operator reference in the calculator.

3. When the calculator wants to *perform* an operation, it invokes a method of the BinaryOperator interface and saves the result in register 1. This method is usually called execute or perform.



Using a BinaryOperator, the calculator can perform a saved operation without using "if":

```
class Calculator {
     private double register1, register2;
     private BinaryOperator operator;

     /** method to perform a binary operation  */
     private void performOperation( ) {
          register1 = operator.perform(register1, register2);
          setDisplayValue( register1 );
     }
```

4. When the user turns the calculator on or presses CLEAR, there is no operator to perform. To avoid "`if (operator != null)...`" in the calculator, define a NullOperator that doesn't do anything. How you implement this depends on how your calculator uses the result of BinaryOperator. For the code example above, we need an *identity* operator.

```
private final BinaryOperator NO_OP = new NullOperator();
/** constructor initializes calculator */
public Calculator {
     clear();
}
/** reset the calculator */
public void clear( )  {
     register1 = regiseter2 = 0;
     operator = NO_OP;
     ...
}
```

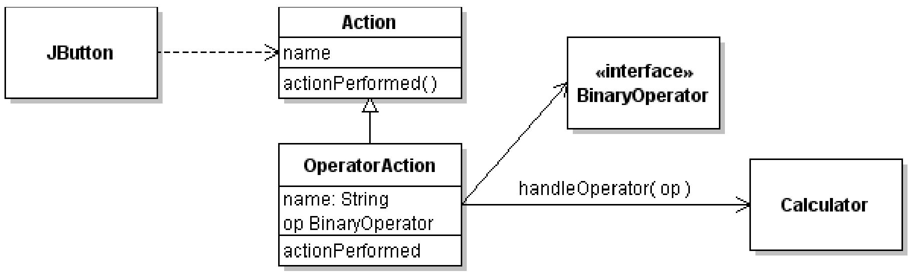
Calculator Method to Set Operator

The `handleOperator` method of the calculator should be changed to accept BinaryOperator objects instead of char. This way, the Calculator won't have to create BinaryOperator objects.

```
public void handleOperator(BinaryOperator op) {
//TODO check state and perform pending operator
   this.operator = op;
}
```

Actions for Operator Keys in User Interface

Use the Command Pattern again -- define Action objects for the binary operations. Each action object invokes handleOperator( ) on the calculator when the user presses a button on the U.I.



The BinaryOperator interface can encapsulate the symbol of the operator (such as "+", "-", "*") making it simple to "program" the Actions. Suppose BinaryOperator has a toString() that returns the key symbol for an operation. OperatorAction can use this to set the text shown on the JButton.

```
class OperatorAction extends AbstractAction {
   private BinaryOperator op;
   public OperatorAction(BinaryOperator op, Calculator calc) {
      super( op.toString() );
      ...
   }
```

Actions for Other Calculator Commands

You can define Actions for all keys on the user interface, so the keypad does not depend directly on the calculator implementation. For example, DigitAction, ClearAction, or EqualsAction.