Classes and Objects in Java – Hiding Information

So far, we have declared all of the fields in our classes in a form similar to that used for variables.

```
class Fraction
{
      int num;
      int den;
}
```

Without specifying the `public` or `private` modifier, these fields are available for viewing or alteration from anywhere in the same directory.

We frequently wish to limit direct access to some fields of a class, so that direct visibility is limited to methods within the same class. To accomplish this, we specify the `private` modifier with those fields that we wish to *hide* from the outside world.

```
class Fraction
{
      private int num;
      private int den;
}
```

Now the fields cannot be seen or altered from outside the `Fraction` class.

Even when limiting *direct* access to a field, we may still wish to allow controlled access to the fields of the object. To accomplish this, we write methods that can be used outside of the class.

```
public int getNumerator()
{
      return this.num;
}
```

Assuming some program was already working with a fraction object named f, the value of the numerator could be retrieved with the statement

```
int n = f.getNumerator();
```

Methods that allow us to gain access to the values in private fields are called *accessor methods*. The form of our example illustrates a common form of accessor methods, using the word "`get`" in conjunction with the desired field.

Similarly, methods that allow us to change the values in private fields are called *mutator methods*.

Creating programs where some information within a class is inaccessible, or hidden, from the outside world is called *encapsulation*. It is possible to encapsulate both data fields and methods, depending upon the use of the `public` and `private` modifiers.

Although accessor and mutator methods may at first seems like extra work for nothing, the encapsulation they provide pays dividends, particularly as programs grow more complex, and as more programmers work on the same projects.

Example 1

The following constructor for the Fraction class takes two parameters, the numerator and denominator of the fraction. By mathematical convention, we do not represent fractions with a negative denominator. This constructor will automatically ensure that a negative denominator will be handled appropriately.

```
public Fraction (int n, int d)
{
      if (d < 0)
      {
            this.num = -n;
            this.den = -d;
      }
      else
      {
            num = n;
            den = d;
      }
}
```

Example 2

The method invert replaces a Fraction object by its inverse, and also maintains the positive denominator convention.

```
public void invert ()
{
      int temp = this.num;
      this.num = this.den;
      this.den = temp;
      if (this.den < 0)
      {
            this.den = -this.den;
            this.num = -this.num;
      }
}
```