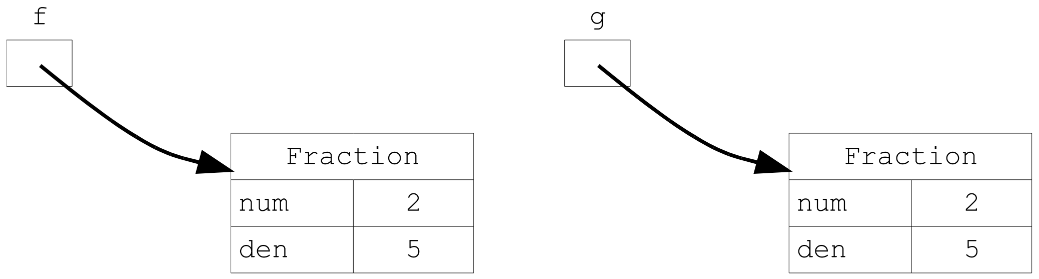
Classes and Objects in Java – Comparing & Displaying Objects

As we have previously discussed, there are significant differences in how we need to work with objects, as compared to the primitive data types.

Given the following declarations,

```
Fraction f = new Fraction(2, 5);
Fraction g = new Fraction(2, 5);
```

we must remember that the variables `f` and `g` act as *references* to the locations in memory containing the objects. A reference is equivalent to an *address* in memory.



Even though the *values* in the *fields* of each object are the same (numerator of 2, denominator of 5), the variables `f` and `g` are *not equal*, because they refer to different locations.

This can be easily illustrated by outputting the value of `f` and `g`:

```
System.out.println(f);
System.out.println(g);
```

which will yield a result similar to

```
Fraction@1cc7c5
Fraction@1d4a32
```

This output tells us that each of the variables f and g are references for a class called “Fraction”. In addition, each of the two objects has a distinct location (address) in memory. Note that these address values will vary according to the conditions of the computer when it runs the code.

Thus a comparison of the two variables, `f` and `g`, should show that they are not the same, and a statement such as

```
System.out.println(f == g);
```

would produce the output “`false`”.

Classes and Objects in Java – Comparing & Displaying Objects

Comparing Objects

In order to compare objects properly, we must test the contents of their fields. This is generally done using an *instance method* similar to the one used with strings, called equals.

```
public boolean equals (Fraction other)
{
     if (this.num == other.num && this.den == other.den)
          return true;
     else
          return false;
}
```

An example of using this method might begin with

```
if (p.equals(q)) ...
```

An equals method does not require all fields to be equal (although this is usually the case). The method can apply whatever criteria we choose for equality. For example, in the case of a `Fraction` class, we might consider that two objects are equal if the ratios of the `num` and `den` fields are equal (which is a more mathematical definition of equality for two fractions).

If we do not write our own equals method, Java supplies a default version for any object we define. Since the default version only uses the == comparison operator, it has limited value to us. To get something more meaningful, we *override* the default method by writing our own method, as shown above.

Displaying Objects

To display values, we have been using the methods `print` and `println`. These methods automatically convert primitive values (from primitive data types such as `int` or `double`) to `String` values for printing. For objects, this conversion is done by a default instance method called `toString`. As previously shown, displaying an object variable using `print` or `println` will produce the identifier of the class and the memory reference of the object.

```
public static void main (String[] args)
{
     Fraction f = new Fraction(2, 3);
     System.out.println(f);
}
```

will produce the output "`Fraction@1cc7c5`".

As with other default instance methods, we can *override* this method with our own definition.

```
public String toString()
{
     return num + "/" + den;
}
```

Now, if we were to run our program, Java would use our custom instance method toString rather than the default. The output from the program would be "`2/3`".