Classes and Objects in Java – Constructors

In creating objects of the type Fraction, we have used statements similar to the following:

```
Fraction f = new Fraction();
```

The parentheses in the expression `Fraction()` makes it look like a method, yet we never created such a method in our class. In fact, it is a special type of method called a *constructor method*. Although we didn't create this method, Java automatically creates a default constructor method. The constructor method has the same identifier as the class (in this case, the word 'Function' is the identifier for both the constructor and the class).

Constructor methods are *instance methods* used to initialize new objects at the time they are created. The default constructor method will perform the following initializations:

1. All numeric fields are set to zero.
2. All `boolean` fields are set to `false`.
3. All reference fields (i.e., fields that refer to an object, rather than a primitive data type) are set to `null`.

Note: The value of `null` is, in essence, 'nowhere'. By setting a reference field to `null`, we are explicitly stating that it does not refer to any object. On the other than, an uninitialized reference field could theoretically contain any value, including locations in memory occupied by running programs. This is one of the many reasons why it is critical to always initialize all variables to known starting values.

Redefining the Constructor Method

Java creates a constructor method by default for all classes. It is possible, however, to specify our own constructor method, which will then replace the default constructor.

Suppose, in the class `Fraction`, we created the following constructor:

```
public Fraction (int n, int d)
{
      this.num = n;
      this.den = d;
}
```

Notice that this constructor method accepts the integer arguments `n` and `d`. We might create a new instance of a `Fraction` object with

```
Fraction f = new Fraction(2, 3);
```

This would declare f to be of type Fraction, create an object of type `Fraction` in memory, set f to refer to that object in memory, and then initialize the object to represent the fraction $\frac{2}{3}$.

Some notes about our first constructor method:

1. The identifier of the constructor is *exactly* the same as the identifier of the class.
2. Unlike every method we have seen so far, the constructor method does not specify a return type (not even `void`).
3. The constructor is an *instance method*, allowing us to use the keyword `this`.
4. The call to the constructor requires the use of the keyword `new`.
5. The parameters were called `n` and `d` to avoid any potential conflicts with `num` and `den` (since the use of the keyword `this` is optional).

Overloading Constructors

Like other methods we have previously discussed, it is also possible to *overload* a constructor method. This allows us to create multiple versions of the constructor method, each with its own signature and each doing some different kind of initialization.

```
// constructor to initialize the new object to be
// equal to an existing object f
public Fraction (Fraction f)
{
      this.num = f.num;
      this.den = f.den;
}
```

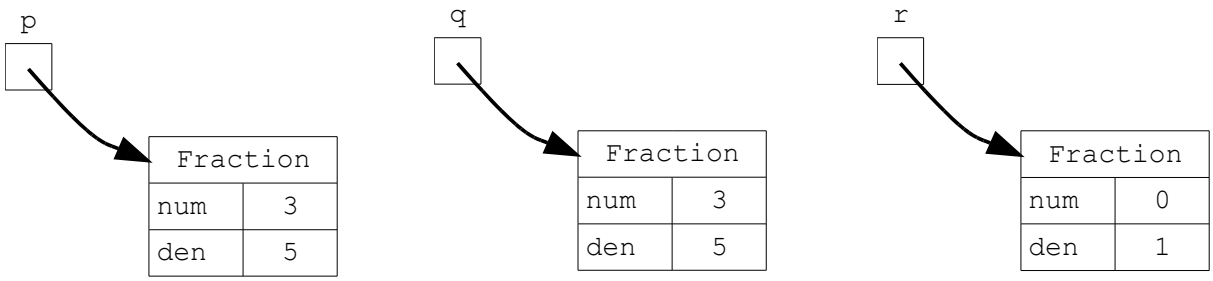
Once we have created our own constructor method, the default constructor method no longer operates. It is possible, however, that we wish to have a constructor method with no parameters, in which case we will have to write our own.

In the case of the Fraction class, the default constructor method will initialize the numerator and denominator to zero, creating the fraction $\frac{0}{0}$. It might be more logical to initialize the fraction to something mathematically safer, such as $\frac{0}{1}$.

```
public Fraction()
{
      this.num = 0;
      this.den = 1;
}
```

Consider the results of the following code fragment:

```
Fraction p = new Fraction(3, 5);
Fraction q = new Fraction(p);
Fraction r = new Fraction();
```



Questions & Exercises

1. Extend the definition of the Fraction class that we have been developing to include the constructors discussed in this section.