Tutorial on Method Overloading

Method overloading is a feature that object-oriented languages like Java provide to programmers.

Method overloading enables programmers to define methods that have the same name, but do slightly different things. Method overloading is most commonly used when defining class constructor methods, but it can be used for any type of instance method or class method.

The advantage of method overloading is that it allows programmers to provide significant flexibility in the classes they design.

Let’s take a look at the use of method overloading when defining Date class constructors first.

```
public class Date
{
      private int month;
      private int day;
      private int year;

      public Date( int m, int d, int y )
      {
            month = m;
            day = d;
            year = y;
      }

      public Date( Date dt )
      {
            month = dt.getMonth();
            day = dt.getDay();
            year = dt.getYear();
      }

      public Date()
      {
            month = 99;
            day = 99;
            year = 9999;
      }

                  …
}
```

Here’s a partial class definition for a `Date` class. Note that three class constructors are defined. This is an example of method overloading. When a programmer creates `Date` objects, Java will try to match the signature used in creating an object to one of the signatures of the constructor methods.

A **signature** is the name of the method and the number and types of arguments.

In this example there are three distinctly different signatures for the class constructors. Notice how much flexibility this gives users of the `Date` class. One can just create a `Date` object and deal with the default values that the default constructor provides, or one can create a `Date` and pass it another `Date` object to initialize it, or one can explicitly specify each component of the `Date` object.

Some examples of creating `Date` objects using these constructors are illustrated below.

```
Date d1 = new Date();

Date d2 = new Date( 10, 10, 2008 );

Date d3 = new Date ( d2 );

Date d4 = new Date ( new Date( 12, 2, 2010 ) );
```

The first example uses the default constructor. The second example passes the `month`, `day` and `year` components as integers. The third example passes an already created `Date` object, and the fourth example creates a `Date` object by combining two different constructors.

In the fourth example, a new `Date` object is created by specifiying `month`, `day`, and `year` components. This new `Date` object is then passed as an argument to the constructor that accepts a `Date` argument. This approach avoids having to define a separate `Date` object that is just used for purposes of initialization.

Now, let's see how method overloading can be applied to some non-constructor methods in the `Date` class. I'm only going to show the skeletons for each method. Here's a few possibilities:

```
public void setDate( int m, int d, int y )
{
                  …
}

public void setDate( String sdt )
{
                  …
}

public void setDate( Date dt )
{
                  …
}
```

In the above examples, I've provided users of the `Date` class with three different ways to set the value of a `Date` object. The first method lets them pass components for a `Date` object. The second method lets them pass a `String` form of `Date` object, such as "10/10/2010". And the third method lets them pass another `Date` object.

Because Java checks argument types when methods are invoked, it can attempt to match the signatures of overloaded methods and select the correct method to execute.

Many of Java's built-in classes offer this kind of flexibility. The challenge for the programmer, however, is to balance flexibility with simplicity. For example, a `Date` class that provided 20 ways to set the value of a `Date` object would be flexible, but it could also be confusing when choosing which way is the best way.