Johns Hopkins University
Introduction to Programming Using Java
605.201

Designing Classes Using Containment

Designing Classes Using Containment

When you design Java classes, one or more of the class members may themselves be objects. For example, consider the Person class below:

```
public class Person
{
  private String firstName;
  private String lastName;
  private Date birthDay;
    …
}
```

The `firstName` and `lastName` members of this class are objects of type `String`, and the `birthDay` member is a `Date` object.

When you include other objects as class members you are using a design technique called containment. In this case, our `Person` class "contains" other objects.

When we create `Person` objects, two `String` objects and a `Date` object will also be created, and the respective class constructors will be invoked.

Constructor Considerations When Using Containment

When you design classes using containment, care must be taken when you define constructors for your classes, because your classes must take responsibility for ensuring that the contained object class constructors are properly invoked.

Let's take a look at a possible partial class definition for the Date class used in the above example.

```
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 String getDateString()
  {
   return month + "/" + day + "/" + year;
  }
   …
}
```

Johns Hopkins University
Introduction to Programming Using Java
605.201

Designing Classes Using Containment

Note that the `Date` class has a parameterized constructor. This means that whenever `Date` objects are created `month`, `day`, and `year` parameters must be supplied. There is no default constructor for the `Date` class.

Now, let's revisit the `Person` class and look at some of its methods.

```
public class Person
{
  private String firstName;
  private String lastName;
  private Date birthDay;

  public String getFirstName()
  {
   return firstName;
  }

  public String getLastName()
  {
   return lastName;
  }

  public String getBirthDayString()
  {
   return birthday.getDateString();
  }

  public Person( String fn, String ln, Date bd )
  {
   firstName = fn;
   lastName = ln;
   birthDay = bd;
  }

  public Person()  // Default constructor
  {
   firstName = "None";
   lastName = "None";
   birthDay = new Date( 99, 99, 9999 );
  }

    …
}
```

This class includes methods to return the `firstName`, `lastName`, and `birthDay` members as `String` objects. It also has two constructors.

The first constructor is a parameterized constructor. It takes `String` arguments for the `firstName` and `lastName`, and a `Date` object for the `birthDay`. The `Person` class members can then be initialized using simple assignment statements.

The second constructor is a default constructor. It will set default values of "none" for the `firstName` and

Designing Classes Using Containment

`lastName` members, and will set a default value for the `birthDay` member. Note that this constructor had to explicitly create a new `Date` object to initialize the birthday member, and three parameters had to be provided because the `Date` class takes a parameterized constructor. Because the `Person` class contains `String` and `Date` objects its constructors must provide a mechanism to make sure the constructors for the contained objects can be properly invoked.

Now, let's look at a program that creates some `Person` objects.

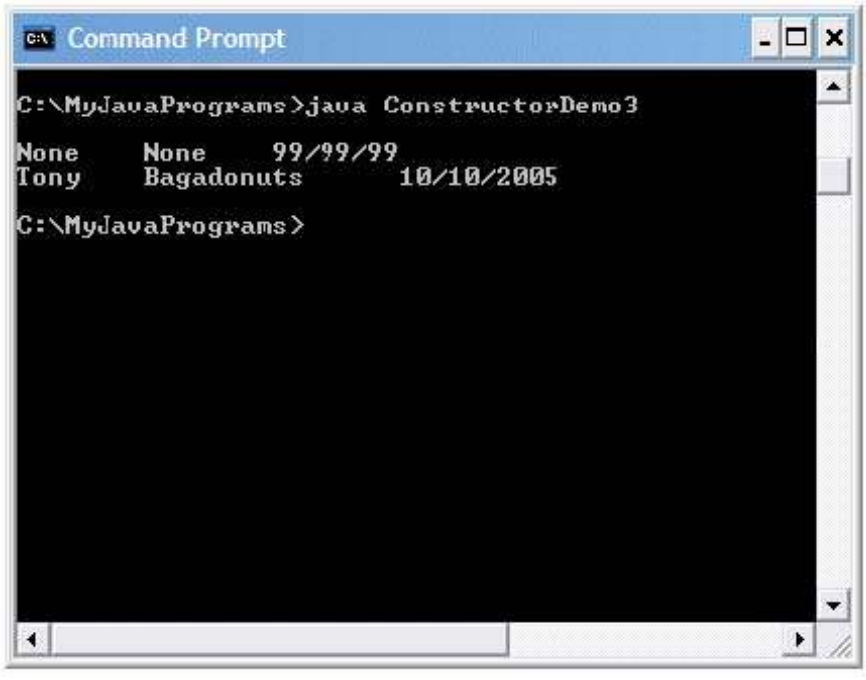
```
public class ConstructorDemo3
{

   public static void main( String [] args )
   {
    Person p1 = new Person();
    Person p2 = new Person( "Tony", "Baggadonuts", new Date(10,10,2005) );

    System.out.println();
    System.out.println( p1.getFirstName() + "\t" + p1.getLastName() + "\t"
                  + p1.getBirthDate() );
    System.out.println( p2.getFirstName() + "\t" + p2.getLastName() + "\t"
                  + p2.getBirthDate() );
   }
}
```

Notice how I provided the `Date` parameter for `Person` object `p2`.

Here's a copy of the program output.

Johns Hopkins University
Introduction to Programming Using Java
605.201

Designing Classes Using Containment

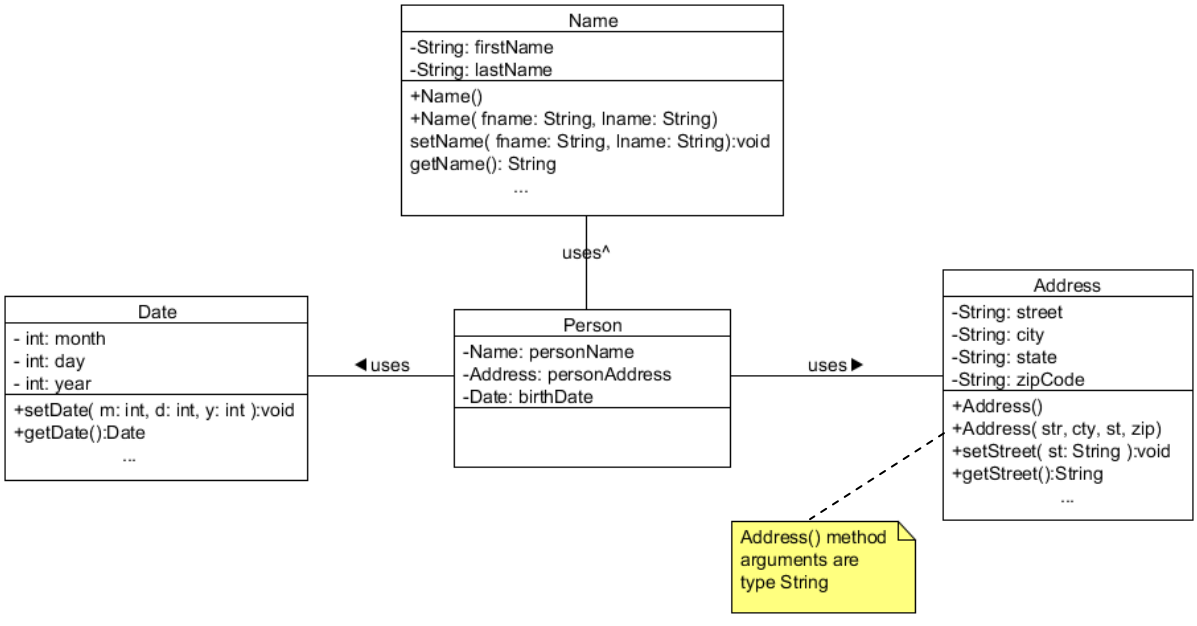


Class Diagram Notation for Class Containment

When we first flush out the design of an application, a Unified Modeling Language (UML) class diagram can be used to show that a design is based on containment. Suppose we had a Name class, Address class, and Date class…and we wanted to reuse those classes to model a Person class. The following class diagram can be used:

Johns Hopkins University
Introduction to Programming Using Java
605.201

Designing Classes Using Containment



When containment is used in a design, we sometimes say a “uses” relationship exists. In this example, the Person class “uses” a Date object, a Name object, and an Address object. (No methods are shown for the Person class for brevity and because the purpose of this diagram is to illustrate the concept of containment.)

Note in the class diagram that Java types are specified for the class attributes and some of the methods. As an example, the setDate() method in the Date class indicates that the method takes three integer arguments: the first argument corresponding to the month, the second corresponding to the day, and the third corresponding to the year. The return type of this method is specified as *void*, so the method does not return a value.
In the Name class, the first two methods are constructors. It is easy to know this because the method names are the same as the class name. Also, by convention, constructors are generally shown as the first methods for a class. Note that the constructors do not have a return type since constructors can’t return a value.

Note the second constructor for the Address class. Only the method parameter names are indicated in the diagram…but not the types of the parameters. The Unified Modeling Language (UML) syntax that we have been using is very flexible regarding how much detail needs to be shown. The more detail that is shown the easier the transition from design to code in general, since we have to make fewer programming decisions. In this example, the parameter types are not shown for two reasons: first, for the sake of brevity; and second, to introduce how comments can be added to a class diagram. In this case, the comment provides information about the parameter types.