UNBC Laboratory Assignment

Computer Science 101—Winter 2008

Factory Methods, Clones, and Automation

Due Date:

This assignment is due Wednesday 2008-02-06.

Purpose:

To extend the Payroll system developed in Laboratory Assignment 2, and thereby gain familiarity with advanced JAVA idioms for object construction.

More particularly, to extend the `Payroll` program so that the addition of new `Employee` subclasses is automated to the point that adding a new subclass of `Employee` is as simple as adding the appropriate `.class` file to the project directory.

Reading Assignment:

Read the entire assignment and Chapter 11 of the textbook before attempting this assignment.

Be sure to understand the answers to the following questions. Discuss them with your lab instructor if you are not sure how they work.

- What advantages does a factory method have over a constructor?
- Is a `clone()` method a factory method?
- Is the `clone()` method of the `Object` class public?
- Can a method that has an `Employee` argument also accept objects of subclasses?
- Can an overriding method declare a `return`-type that is a superclass of the overridden method?
- Can an overriding method declare a `return`-type that is a subclass of the overridden method?

⇒ Hand in answers to these questions attached to your lab assignment. (See the "Summary" section on page 4.)

Copy constructors:

A copy constructor is a constructor whose sole argument is another object of the same class. Although not as important as in C++, copy constructors greatly simplify the operation of creating copies of objects.

⇒ If you haven't already done so, add copy constructors to the `Employee` class and each of its subclasses.

Simple `clone()` operations

An important part of being able to generate objects from a factory method is the ability to clone a prototype object of a class. With copy constructors in place, this is easy to implement. For instance in `Contract` the clone function can be written as

```
public Contract clone () { return new Contract(this) ; }
```

⇒ Add `.clone()` methods to each of `Employee` and its subclasses.

(Note that we are overriding the "`protected Object clone() throws CloneNotSupported`" method of the `Object` class.)

Employee Factories

A factory method is static method of a class that returns objects of that class, or possibly of its subclasses. This overcomes a shortcoming of constructors, which can only ever return objects of the class that they are members of.

In particular, we want to add to the `Employee` class a method with signature

```
public static Employee getInstanceFor(EmployeeCode ec) { ... }
```

Although the return type is `Employee` we want the method to return an object of a subclass whose `.getCode()` method returns the `EmployeeCode ec` argument.

In order to initialize the connection between various `EmployeeCodes` and `Employee` classes we also want to have a method

```
protected static void     register(Employee e) { ... }
```

We expect that the argument to this method will in fact be an object of a subclass of Employee. By using `e.getCode()` the `register` function can associate this subclass object `e` with the appropriate `EmployeeCode`.

In order to do this in an automatated way, we need to have a data structure that relates `EmployeeCodes` to `Employees`. The abstract `java.util.Map<K,V>` class and the concrete instantiation `java.util.TreeMap<K,V>` can be used to do this.

In the `Employee` class we need to add

```
private static java.util.Map<EmployeeCode,Employee> registrations
    = new  java.util.TreeMap<EmployeeCode,Employee>() ;
```

and methods

```
protected static void     register(Employee e) { ... }
public    static Employee getInstanceFor(EmployeeCode ec) { ... }
```

The `void register(Employee e)` method should register an instance of a subclass against its `EmployeeCode`. If the `EmployeeCode` has already been registered against another class, the program should take some emergency action.

The `Employee getInstanceFor(EmployeeCode ec)` method should return a `clone()` of the registered instance for the `EmployeeCode ec` if there is one, otherwise it should either return `null` or an instance of an appropriate `Unknown` class.

These two methods will likely use the pre-existing `Map<EmployeeCode,Employee>` methods

```
    public boolean  containsKey(EmployeeCode ec) ;
    public Employee get(EmployeeCode ec) ;
    public void     put(EmployeeCode ec, Employee e) ;
```

Look at the online documentation for these methods if their meaning does not seem clear.

⇒ Write and add the

- the `registrations` field,
- the `register` method, and
- the `getInstanceFor` method

to the `Employee` class.

⇒ Test your `Employee` factory methods.

Note that by using logic like that shown in Figure 2 on page 7 the only code that now explictly needs to be modified to add a new class is the `loadClasses` method.

Class `static` initialization

The `boolean classesLoaded` in Figure 2 is bit worrisome. In fact, JAVA provides a method for ensuring that a piece of code is executed once per class per program. This is called a static initialization block and is shown in Figure 3 on page 8.

However, it is even more natural to separate this static initialization into the subclasses of the `Employee` class. For instance in the `Contract` class, have the code

```
  static { register(new Contract()) ; }
```

⇒ Add `static` initialization blocks to each of the subclasses of the `Employee` class, and remove the same initialization from other places that it may have been.

Auto-loading all of the classes in a directory

This almost completely automates the addition of classes. Unfortunately, these `static`-blocks are not executed until the class is loaded, and the class is not loaded until it is mentioned.

To force the autoloading of all of the classes in the runtime directory of the program, you need to use some additional trickery like that shown in Figure 1 on page 6. This method loads all of the `.class`-files in a directory[1].

Here is a brief explanation of the method. The `String [] list()` method of `java.io.File` produces a list of all of the file names in a directory. The `void forName(String)` static method of the `Class` class loads a class with a given name. The remainder of the code searches for strings that end with `.class` and removes that part of the string to get a class name.

To use this method, you can call it as shown:

```
loadEmployeeClasses(new File("/home/casper/Code/Java/payroll")) ;
```

assuming that `/home/casper/Code/Java/payroll` is where the appropriate class files lie.

⇒ Implement autoloading, and test it with your current classes.

⇒ Implement a `Commission` class that meets the description below. Compile your `Commission.java` file, and *without making any other changes to your code* see if your program can now handle `Commission` employees.

Description of the Commission Employees

Commission Commissioned employees work 37.5 hours per week for a varying hourly rate. In addition they are payed a commission on sales, and possibly other expenses related to travel costs. A typical commission employee input record looks like

```
CM 19.31 4000.00 150.00
```

indicating an employee that earns $19.31 per hour, and had commissions totalling $4000.00 and other expenses totalling $150.00.

Deductions for commissioned employees are the same as for part time employees on hourly wages, together with a flat 15% on commissions earned. There are no deductions on the other payments.

Summary

Congratulations!

If you have made it this far, you have created a JAVA program that can be modified to handle arbitrary new classes of `Employees` *solely by adding* `.class`-files for those new classes.

At this point you should feel that you have a good understanding of what distinguishes Object Oriented Programming from other programming methodologies.

Hand in

- answers to the questions at the beginning;
- a listing of program without the `Commission` class attached, showing that it compiles and runs, and that `CM` employee codes are dealt with as errors; and

[1]This code assumes that they are part of the anonymous package. If you have added `package` statements to your classes, you need to modify this slightly.

- show how attaching a `Commission` class causes the program to run differently.

The remaining section is completely optional.

[OPTIONAL] Fancy `clone()` operations

The `Object.clone` method gives a fast way to create shallow copies. However, the technique is somewhat more complicated than the one given above:

1. First, the `Employee` class must implement the interface `Cloneable`. As this interface has no actual methods, this is simply a matter of adding "`implements Cloneable`" to the `Employee` class declaration.
2. Secondly, the `Employee` `clone()` method needs to handle the `CloneNotSupported` exception even though it will never happen! The cleanest way to do this is to write something like

```
public Employee clone () {
    Employee e = null ;
    try {
          e = (Employee) (super.clone());
      } catch (CloneNotSupportedException ex) {}
    return e ;
}
```

3. Once `Employee` declares that it `implements Cloneable` and has a `clone()` function that doesn't threaten to throw `CloneNotSupportedException`, the clone functions in the derived classes can be as simple as (in `Contract`)

```
public Contract clone () { return (Contract) (super.clone()) ; }
```

4. For subclasses that have mutable fields, in order to implement a deep clone the code should be written more as (in `Contract`):

```
public Contract clone()
    {
    Contract c = (Contract)(super.clone()) ; // shallow copy
    c.mutableField1 = c.mutableField1.clone() ; // copy fields deeply
    c.mutableField2 = c.mutableField2.clone() ;
    }
```

5. You *cannot* mix this technique with the previous technique.

⇒ [optional]
Implement `.clone()` operations using the `Object` class protected method.

```
public static void loadEmployeeClasses(File directory)
    throws ClassNotFoundException
{
String[] classNames =  directory.list() ;
final String tail = ".class" ;
final int tailSize = tail.length() ;
for (int i = 0, ell = classNames.length; i < ell; ++i)
    {
    String x = classNames[i] ;
    if (x.endsWith(tail))
        Class.forName(x.substring(0, x.length()-tailSize));
    }
}
```

Figure 1: `loadFiles`

```
// various import's
public class Bob {
  private static boolean classesLoaded = false ;
  public static void loadClasses()
      {
      if (!classesLoaded)
          {
          Employee.register(new Parttime()) ;
          Employee.register(new Fulltime()) ;
          Employee.register(new Contract()) ;
          classesLoaded = true ;
          }
      }

  public static void processFiles(Reader in, Writer out)
      {
      loadClasses() ;
      Report report = new Report(out);
      report.setLinesPerPage(50) ;
      processReport(new Scanner(in), report) ;
      report.close() ;
      return ;
      }

  public static void processReport(Scanner sin, Report report)
      {
      if (!sin.hasNext()) return;
      EmployeeCode code = new EmployeeCode(sin.next()) ;
      Employee current = Employee.getInstanceFor(code) ;
      current.readFrom(sin) ;

      while (true)
          {
          report.print(current) ;
          if (!sin.hasNext()) break ;
          code = new EmployeeCode(sin.next()) ;
          current = Employee.getInstanceFor(code) ;
          current.readFrom(sin) ;
          }
      return ;
      }
}
```

Figure 2: `processFiles` version 2

```
// various import's
public class Bob {
  static
      {
      Employee.register(new Parttime()) ;
      Employee.register(new Fulltime()) ;
      Employee.register(new Contract()) ;
      }

  public static void processFiles(Reader in, Writer out)
      {
      Report report = new Report(out);
      report.setLinesPerPage(50) ;
      processReport(new Scanner(in), report) ;
      report.close() ;
      return ;
      }

  public static void processReport(Scanner sin, Report report)
      {
      if (!sin.hasNext()) return;
      EmployeeCode code = new EmployeeCode(sin.next()) ;
      Employee current = Employee.getInstanceFor(code) ;
      current.readFrom(sin) ;

      while (true)
          {
          report.print(current) ;
          if (!sin.hasNext()) break ;
          code = new EmployeeCode(sin.next()) ;
          current = Employee.getInstanceFor(code) ;
          current.readFrom(sin) ;
          }
      return ;
      }
}
```

Figure 3: processFiles version 3