UNBC Laboratory Assignment
Computer Science 101—Winter 2013

Factory Methods, Clones, and Automation

Due Date:

This assignment is *optional*. Please submit by Friday 2013-04-05.

Purpose:

To extend the Payroll system developed in Laboratory Assignment 4, and thereby gain familiarity with advanced Java idioms for object construction and polymorphism.

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.

Clone functions:

One of the disadvantages of copy constructors (and constructors in general), is that they cannot provide run-time polymorphism through overriding. On the other hand you *can* ask an object to make a copy of itself in a polymorphic way. This is the purpose of a `.clone()` method.

One generic technique for creating `.clone()` functions is to build the clone function from a copy constructor.[1]

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

(Note that the `.clone()` method is an override of a method in the `Object` class, despite the change in return type.)

Exercise 2

⇒ Add copy constructors and `.clone()` functions to the `Employee` class and each of the `Employee` subclasses of Laboratory Assignment 4.

[1] Another more specifically Java-based technique is to implement the `Cloneable` interface and use `super.clone()` to make a shallow copy, and then add whatever code is needed to provide a deep copy. This is technically tricky.

```
public class Contract
{
    private static Contract adam ;

    public static Contract getInstance()
        {
            if (adam==null)
                adam = new Contract () ;
            return adam ;
        }
    // ...
}
```

Figure 1: `static` object generating method

```
          if      (code.equals("FT"))
              current = Fulltime.getInstance().clone() ;
          else if (code.equals("PT"))
              current =  Parttime.getInstance().clone() ;
          else if (code.equals("CO"))
              current = Contract.getInstance().clone() ;
          else
              {
                  current = UnclassifiedEmployee
                    .getInstance().clone().setCode(code) ;
              }
```

Figure 2: Modified object choice logic

⇒ In preparation for the following parts of this assignment, add a *static* method to each subclass that returns an object of that class. The code can look something like that shown in Figure 1.

⇒ Now change the `Employee` generating logic to look like Figure 2.

⇒ Show that your modified code works at this stage.

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(String employeeCode) { ... }
```

Although the return type is `Employee` we want the method to return an object of a subclass that corresponds to the `employeeCode` given.

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

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

We expect that the argument to this method will in fact be an object of a subclass of Employee. Although we won't use the `register` method in quite this way, one can imagine writing

```
private static void setupFactory()
    {
        register("CO", Contract.getInstance()) ;
        register("FT", Fulltime.getInstance()) ;
        register("PT", Parttime.getInstance()) ;
    }
```

In order to do this in an automatated way, we need to have a data structure that relates `Stringss` 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<String,Employee> registrations
    = new  java.util.TreeMap<String,Employee>() ;
```

and methods

```
protected static void     register(String, Employee e) { ... }
public    static Employee getInstanceFor(String employeeCode) { ... }
```

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.

```
import java.util.Map;
import java.util.TreeMap;

public class Employee {

  // Registration related static stuf

  private static Map<String,Employee> registrations
      = new  TreeMap<String,Employee>() ;

  protected static void register(String code, Employee employee)
      {
      // throwing an exception would be even better!
      assert (!registrations.containsKey(code)) ;
      registrations.put(code,employee) ;
      }

  // a factory method for subclass instances.
  // relies on the registrations to be filled in by the time that
  // it is used.
  public static Employee getInstanceFor(EmployeeCode ec)
      {
      return
          registrations.containsKey(ec)
          ? registrations.get(ec).clone()
          : new Unknown(ec) ;
      }
// ....
}
```

Figure 3: Java code to register employees

```
code = employeeScanner.next() ;
current = getInstanceFor(code).clone() ;
current.readFrom(employeeScanner) ;
```

Figure 4: Further modified object choice logic (See Figure 2).

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.

These can be written as shown in Figure 3. Note that that at this point the code in Figure 2 can be simplified to that shown in Figure 4 (assuming that you call `register` before hand.)

Exercise 3

⇒ Write and add the

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

to the `Employee` class.

⇒ Test your `Employee` factory methods.

Class `static` initialization

One question that remains is where to put the calls to `register` each class. The correct answer is to have each class register itself. JAVA has a mechanism to do this. A top level **block** in a class preceded by the word `static` is executed when the class is loaded.

For instance in the `Contract` class, the code

```
public class Contract {
...
  static { register("CO", Contract.getInstance()) ; } // <--------------
}
```

would cause the `Contract` class to be registered as soon as it is loaded. The only remaining hurdle is causing all classes to be loaded.

⇒ 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 5 on the following page. This method loads all of the `.class`-files in a directory[2].

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.

Exercise 4

⇒ 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.

[2]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.

```
1   public static void loadEmployeeClasses(File directory)
2       throws ClassNotFoundException
3   {
4   String[] classNames =  directory.list() ;
5   final String tail = ".class" ;
6   final int tailSize = tail.length() ;
7   for (int i = 0, ell = classNames.length; i < ell; ++i)
8       {
9       String x = classNames[i] ;
10      if (x.endsWith(tail))
11          Class.forName(x.substring(0, x.length()-tailSize));
12      }
13  }
```

Figure 5: `loadFiles`

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

- listings for each of the exercises above, so that the stages of implementation can be seen.