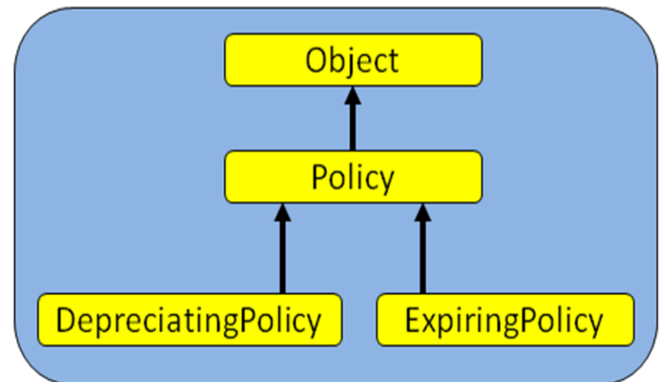
COMP1406 - Assignment #4

(Due: **Mon. Mar 5th @ 12 noon**)

This assignment builds a very simple insurance company system. You will gain practice with inheritance, overriding methods, an abstract class and abstract methods. The insurance system will keep track of the insurance company's clients. Each client will have at least one policy with the company. The last part of the assignment is unrelated ... and involves creating a simple GUI window. **Changes have been made (shown in yellow) on Feb 15th.**

(1) **Policy** Classes

We will be creating this class hierarchy:



To begin, copy the code from the following **Policy** class:

```
public class Policy {
    private static int NEXT_POLICY_NUMBER = 1;

    private     int     policyNumber;
    protected   float   amount;

    public Policy(float amt) {
        amount = amt;
        policyNumber = NEXT_POLICY_NUMBER++;
    }

    public int getPolicyNumber() { return policyNumber; }
    public float getAmount() { return amount; }

    public String toString() {
        return String.format("Policy: %04d amount: $%1.2f", policyNumber, amount);
    }

    public boolean isExpired() {
        return false;
    }
}
```

Now define a **DepreciatingPolicy** class as a subclass of **Policy** which has the following:

- a **private** instance variable called rate of type **float** which represents the rate at which the policy depreciates each time a claim is made. For example, a rate of 0.10 means that the value of the policy (i.e., the amount) depreciates 10% each time a claim is made (we will discuss the making of claims in the next section).
- a **public** constructor which takes a **float** representing the amount and another **float** representing the rate. This constructor **MUST** make full/proper use of inheritance.
- a **public** get method for the rate.
- a **toString()** method that returns a **String** with the following example format:
  `DepreciatingPolicy: 0001 amount: $320.00 rate: 10.0%`

  You **MUST** make use of inheritance by calling the **toString()** from the superclass. Also, the rate must be properly formatted.

- an instance method called **isExpired()** which returns **true** if the amount of this policy has depreciated to **0** (assume a value less than 0.01 is zero). You **MUST** write this method without any IF/SWITCH/TERNARY conditional statements.
- an instance method called **depreciate()** which reduces the amount of the policy by the rate percentage. For example, if the amount is **$100** and the rate is **0.10**, then the amount after this method is called should be **$90**.

Define an **ExpiringPolicy** class as a subclass of **Policy** which has the following:

- a **private** instance variable called expiryDate of type **Date** (from **java.util.Date** package) that contains the date after which the policy is invalid.
- a **public** constructor which takes a **float** representing the amount and a **Date** representing the expiryDate. This constructor **MUST** make full/proper use of inheritance.
- a **public** constructor which takes a **float** representing the amount. This constructor **MUST** make full/proper use of inheritance. The expiryDate should then be set to exactly one year after the policy was created. Here is how you can do this:

```
GregorianCalendar aCalendar = new GregorianCalendar();
aCalendar.add(Calendar.YEAR,1);
expiryDate = aCalendar.getTime();
```

- a **public** get method for the expiryDate.
- a **toString()** method that makes use of inheritance by calling the **toString()** from the superclass and returns a **String** with the following example format which includes the parentheses (i.e., look at the **SimpleDateFormat** class described in section 13.3 of the course notes in order to see how to format the date properly). Notice the difference (underlined) between expired and no-expired policies:

```
ExpiringPolicy: 0001 amount: $320.00 expired on: April 30, 2001 (12:08PM)
ExpiringPolicy: 0006 amount: $500.00 expires: May 23, 2023 (4:46AM)
```

- An instance method called **isExpired()** which returns **true** if the computer's current date is ON or AFTER the expiryDate and **false** otherwise. (Hint: the **Date** class has methods **before(Date d)** and **after(Date d)** ... see section 13.3 of the course notes).

Now test your new classes with the following program:

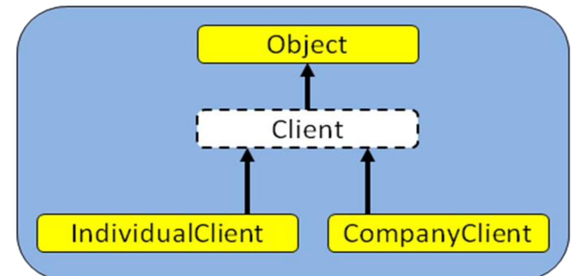
```
import java.util.*;

public class PolicyTester {
    public static void main(String args[]) {
        System.out.println(new Policy(320));
        DepreciatingPolicy p1 = new DepreciatingPolicy(500.1f, 0.1f);
        System.out.println(p1);
        p1.depreciate();
        System.out.println(p1);
        System.out.println(new ExpiringPolicy(1000));
        Date expDate = new GregorianCalendar(2021, 0, 2, 23, 59).getTime();
        ExpiringPolicy p2 = new ExpiringPolicy(2000, expDate);
        System.out.println(p2);
        System.out.println(p2.isExpired());
        expDate = new GregorianCalendar(2013, 3, 1, 23, 59).getTime();
        ExpiringPolicy p3 = new ExpiringPolicy(2000, expDate);
        System.out.println(p3);
        System.out.println(p3.isExpired());
    }
}
```

Here is the expected output (your date on line 4 will differ):

```
Policy: 0001 amount: $320.00
DepreciatingPolicy: 0002 amount: $500.10 rate: 10.0%
DepreciatingPolicy: 0002 amount: $450.09 rate: 10.0%
ExpiringPolicy: 0003 amount: $1000.00 expires: May 16, 2018 (01:18PM)
ExpiringPolicy: 0004 amount: $2000.00 expires: January 02, 2021 (11:59PM)
false
ExpiringPolicy: 0005 amount: $2000.00 expired on: April 01, 2013 (11:59PM)
true
```

---

(2) The **Clients**



We will now define the following hierarchy:

To begin, copy the code from the following **Client** class:

```
import java.util.*;

public abstract class Client {
    private static final int MAX_POLICIES_PER_CLIENT = 10;

    private static int NEXT_CLIENT_ID = 1;

    private   String         name;
    private   int            id;
    protected Policy[]       policies;
    protected int            numPolicies;

    public Client(String n) {
        name = n;
        id = NEXT_CLIENT_ID++;
        policies = new Policy[MAX_POLICIES_PER_CLIENT];
        numPolicies = 0;
    }

    public String getName() { return name; }
    public int getId() { return id; }
    public Policy[] getPolicies() { return policies; }
    public int getNumPolicies() { return numPolicies; }

    public String toString() {
        return String.format("Client: %06d amount: %s", id, name);
    }
}
```

Create the following in the **Client** class:

- a **public** method called totalCoverage() which returns a **float** containing the total amount of coverage of all the client's policies.
- a **public** method called addPolicy(**Policy** p) which adds the given **Policy** to the array of policies, provided that the limit has not been reached ... and returns the **Policy** object, or **null** if not added.
- a **public** method called openPolicyFor(**float** amt) which creates a new **Policy** for the amount specified in the parameter and adds it to the client using (and returning the result from) the **addPolicy()** method

- a **public** method called openPolicyFor(**float** amt, **float** rate) which creates a new **DepreciatingPolicy** for the amount and rate specified in the parameters and adds it to the client using (and returning the result from) the **addPolicy()** method.
- a **public** method called openPolicyFor(**float** amt, Date expire) which creates a new **ExpiringPolicy** for the amount and expiry date specified in the parameters and adds it to the client using (and returning the result from) the addPolicy() method above.
- a **public** method called getPolicy(**int** polNum) which examines all **Policy** objects in policies. If one is found whose policyNumber matches the input parameter, then it should be returned by the method. Otherwise **null** is returned.
- a **public** method called cancelPolicy(**int** polNum) which removes the police from the client's list with the given policy number, if found. It should return true if the policy was found, otherwise it should return false. When a policy is removed from the array, the empty location in the array should be replaced by the last policy in the array.
- a **public abstract** method called makeClaim(**int** polNum) that returns a **float.**

Now create the two subclasses of **Client**, one called **IndividualClient** and the other called **CompanyClient**. Try compiling them. Notice that your test code will not compile. That's because subclasses DO NOT inherit constructors and java requires a constructor to be available. Do the following:

- Create a constructor (taking a single String argument) for each of these subclasses that calls the one in **Client**.
- Go back and alter your **toString()** method in **Client** class so that the proper client type is displayed. For example: `CompanyClient 0001: Bob B. Pins` To do this, you will want to figure out how to get the name of the class as a String. (hint: type `this.` and then let IntelliJ show a list of available methods. You need to find out what class you have and then find out the name of the class).
- Implement the makeClaim(**int** polNum) method in the **IndividualClient** class. If the policy with that number is not found or has expired, then nothing is to be done, just return **0**. **IndividualClients** are allowed to make only 1 claim per regular **Policy** but **DepreciatingPolicy**s and **ExpiringPolicy**s can have multiple claims. So, you should cancel a regular policy after a claim is made on it. If the claim is being made for a **DepreciatingPolicy**, then you must make sure to depreciate the policy. This method should return the amount of the policy (amount after depreciating in the case of **DepreciatingPolicy**s) if the claim was made ok, otherwise 0 is returned.
- In the **Policy** class, implement a method called handleClaim() that returns the amount of the policy.
- In the **DepreciatingPolicy** class, implement handleClaim() so that it returns the amount of the policy but also depreciates the policy. Note that the amount returned is the amount of the policy before it was depreciated.
- In the **ExpiringPolicy** class, implement handleClaim() so that it returns the amount of the policy as long as the policy has not yet expired, otherwise it returns 0.
- In the **CompanyClient** class, implement the makeClaim(**int** polNum) method so that it first checks to make sure the policy is one belonging to this client and then uses the double-dispatching technique by calling the handleClaim() method. You **MUST NOT** use any IF statements to determine the policy type ... that's would undo the advantages of double-dispatching. Note that **CompanyClients** DO NOT have their policy removed afterwards, and so you should make use of the getPolicy() method. If no policy is found with the given **polNum**, return 0;

(3) Testing

Now test what you have done with the following program:

```
import java.util.GregorianCalendar;

public class ClientTester {
    public static void main(String args[]) {
        // Create an individual client, open some policies and then make some claims
        IndividualClient ic = new IndividualClient("Bob B. Pins");
        ic.openPolicyFor(100);
        ic.openPolicyFor(200, 0.10f);
        ic.openPolicyFor(300, new GregorianCalendar(2020, 0, 2, 23, 59).getTime());
        ic.openPolicyFor(400, new GregorianCalendar(2009, 5, 4, 12, 00).getTime());
        Policy p = new Policy(500);

        System.out.println("Here are the Individual Client's policies:");
        for (int i=0; i<ic.getNumPolicies(); i++)
            System.out.println("  " + ic.getPolicies()[i]);

        System.out.println("Making claims:");
        System.out.println(String.format("  Claim for policy 0001: $%6.2f",ic.makeClaim(1)));
        System.out.println(String.format("  Claim for policy 0002: $%6.2f",ic.makeClaim(2)));
        System.out.println(String.format("  Claim for policy 0003: $%6.2f",ic.makeClaim(3)));
        System.out.println(String.format("  Claim for policy 0004: $%6.2f",ic.makeClaim(4)));
        System.out.println(String.format("  Claim for policy 0005: $%6.2f",ic.makeClaim(5)));

        System.out.println("Here are the Individual Client's policies after claims:");
        for (int i=0; i<ic.getNumPolicies(); i++)
            System.out.println("  " + ic.getPolicies()[i]);

        System.out.println(String.format("The total policy coverage for this client: $%1.2f",
                           ic.totalCoverage()));

        // Create a company client, open some policies and then make some claims
        CompanyClient cc = new CompanyClient("The Pillow Factory");
        cc.openPolicyFor(1000);
        cc.openPolicyFor(2000, 0.10f);
        cc.openPolicyFor(3000, new GregorianCalendar(2020, 0, 2, 23, 59).getTime());
        cc.openPolicyFor(4000, new GregorianCalendar(2009, 5, 4, 12, 00).getTime());

        System.out.println("\nHere are the Company Client's policies:");
        for (int i=0; i<cc.getNumPolicies(); i++)
            System.out.println("  " + cc.getPolicies()[i]);

        System.out.println("Making claims:");
        System.out.println(String.format("  Claim for policy 0006: $%7.2f",cc.makeClaim(6)));
        System.out.println(String.format("  Claim for policy 0007: $%7.2f",cc.makeClaim(7)));
        System.out.println(String.format("  Claim for policy 0008: $%7.2f",cc.makeClaim(8)));
        System.out.println(String.format("  Claim for policy 0009: $%7.2f",cc.makeClaim(9)));
        System.out.println(String.format("  Claim for policy 0005: $%7.2f",cc.makeClaim(5)));

        System.out.println("Here are the Company Client's policies after claims:");
        for (int i=0; i<cc.getNumPolicies(); i++)
            System.out.println("  " + cc.getPolicies()[i]);

        System.out.println(String.format("The total policy coverage for this client: $%1.2f",
                           cc.totalCoverage()));

        System.out.println("Cancelling policy #12 ... did it work: " + cc.cancelPolicy(12));
        System.out.println("Cancelling policy #8 ... did it work: " + cc.cancelPolicy(8));

        System.out.println(String.format("The total policy coverage for this client: $%1.2f",
                           cc.totalCoverage()));
    }
}
```

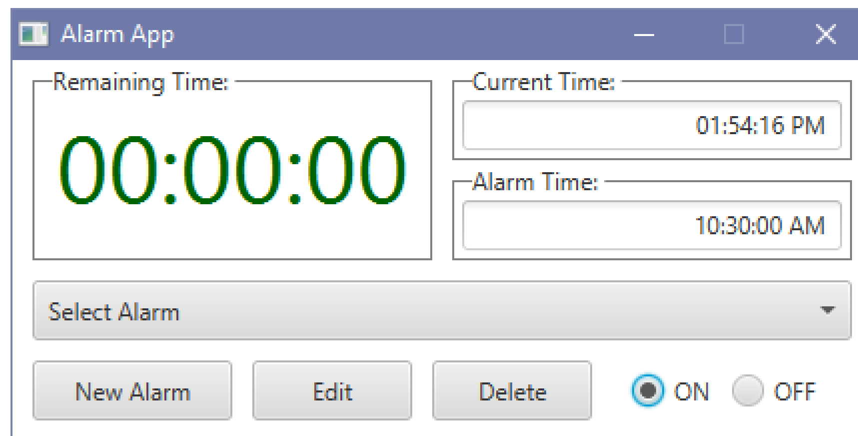
Here is the expected output:

```
Here are the Individual Client's policies:
  Policy: 0001 amount: $100.00
  DepreciatingPolicy: 0002 amount: $200.00 rate: 10.0%
  ExpiringPolicy: 0003 amount: $300.00 expires: January 02, 2020 (11:59PM)
  ExpiringPolicy: 0004 amount: $400.00 expired on: June 04, 2009 (12:00PM)
Making claims:
  Claim for policy 0001: $100.00
  Claim for policy 0002: $180.00
  Claim for policy 0003: $300.00
  Claim for policy 0004: $   0.00
  Claim for policy 0005: $   0.00
Here are the Individual Client's policies after claims:
  ExpiringPolicy: 0004 amount: $400.00 expired on: June 04, 2009 (12:00PM)
  DepreciatingPolicy: 0002 amount: $180.00 rate: 10.0%
  ExpiringPolicy: 0003 amount: $300.00 expires: January 02, 2020 (11:59PM)
The total policy coverage for this client: $880.00

Here are the Company Client's policies:
  Policy: 0006 amount: $1000.00
  DepreciatingPolicy: 0007 amount: $2000.00 rate: 10.0%
  ExpiringPolicy: 0008 amount: $3000.00 expires: January 02, 2020 (11:59PM)
  ExpiringPolicy: 0009 amount: $4000.00 expired on: June 04, 2009 (12:00PM)
Making claims:
  Claim for policy 0006: $1000.00
  Claim for policy 0007: $2000.00
  Claim for policy 0008: $3000.00
  Claim for policy 0009: $    0.00
  Claim for policy 0005: $    0.00
Here are the Company Client's policies after claims:
  Policy: 0006 amount: $1000.00
  DepreciatingPolicy: 0007 amount: $1800.00 rate: 10.0%
  ExpiringPolicy: 0008 amount: $3000.00 expires: January 02, 2020 (11:59PM)
  ExpiringPolicy: 0009 amount: $4000.00 expired on: June 04, 2009 (12:00PM)
The total policy coverage for this client: $9800.00
Cancelling policy #12 ... did it work: false
Cancelling policy #8 ... did it work: true
The total policy coverage for this client: $6800.00
```

---

(4) A GUI

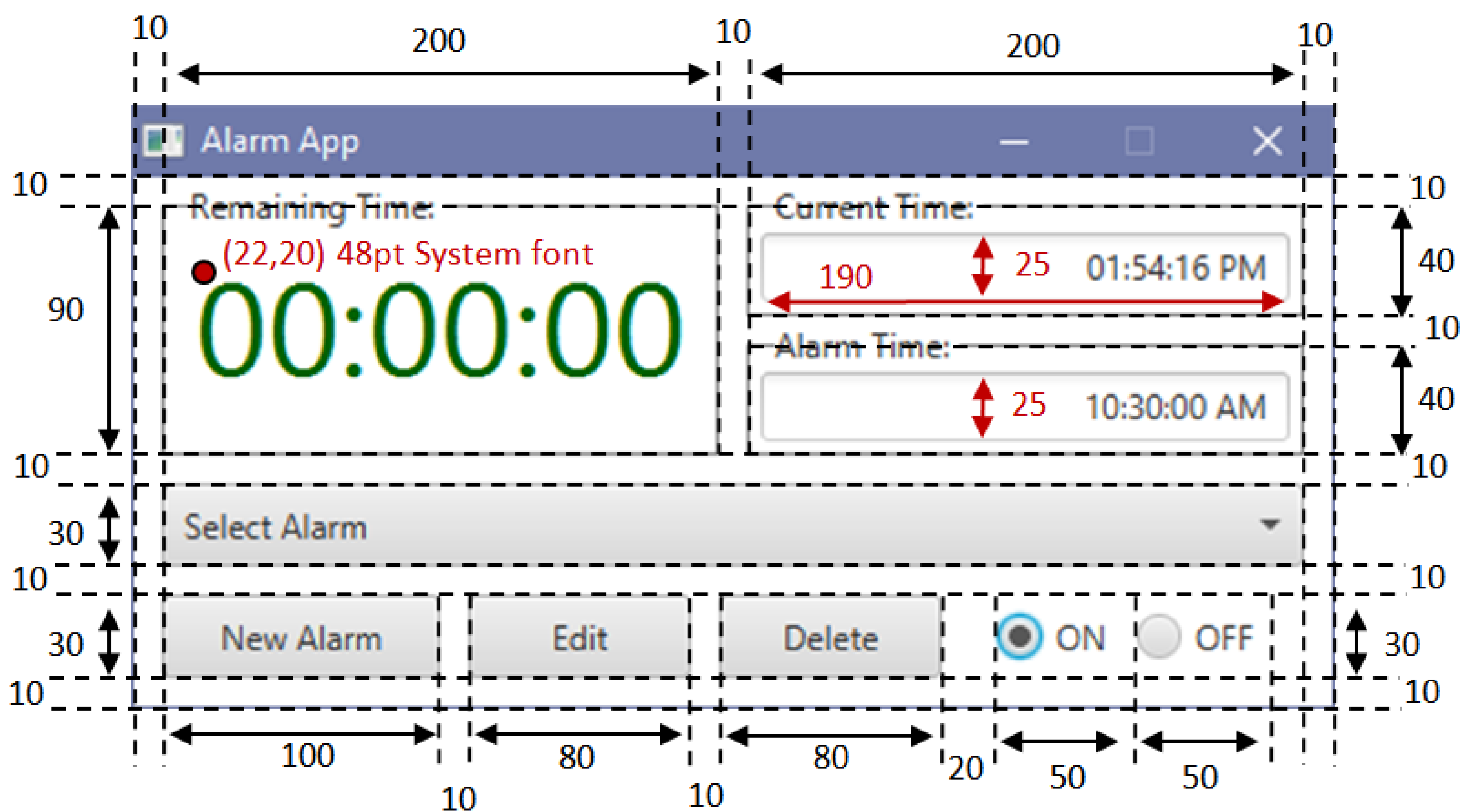
Define a class called **AlarmApp** that creates the following window by laying the components out manually using **relocate()** and **setPrefSize()**:



Note that the GUI MUST contain the following:

- a white background **Pane** (i.e., `"-fx-background-color: white;"`),
- three inner **Pane** objects with borders (i.e., `"-fx-background-color: white; -fx-border-color: gray;  -fx-padding: 4 4;"`),
- **Labels** on those bordered panes (i.e., `"-fx-background-color: white; -fx-translate-y: -8; -fx-translate-x: 10;"`),
- a large (e.g., 48px or anything reasonable) system font **Label** with a dark green color in the "Remaining Time" pane,
- right-aligned **TextFields** in the Current Time and Alarm Time panes with the given times shown,
- a **ComboBox** with prompt text "Select Alarm" and it should contain 3 entries: "Weekday", "Saturday" and "Sunday",
- three **Buttons**,
- and two **RadioButtons** belonging to the same **ToggleGroup**.

The dimensions of everything is as follows:



IMPORTANT SUBMISSION INSTRUCTIONS:

Submit your ZIPPED **IntelliJ project file that contains all your .java files** as you did during the first tutorial for assignment 0.

- YOU WILL LOSE MARKS IF YOU ATTEMPT TO USE ANY OTHER COMPRESSION FORMATS SUCH AS **.RAR**, **.ARC, .TGZ**, **.JAR**, **.PKG**, **.PZIP**, etc.

- If your internet connection at home is down or does not work, we will not accept this as a reason for handing in an assignment late ... so make sure to submit the assignment WELL BEFORE it is due !

- You WILL lose marks on this assignment if any of your files are missing. So, make sure that you hand in the correct files and version of your assignment.   You will also lose marks if your code is not written neatly with proper indentation.   See examples in the notes for proper style.