Teacher Notes for Design Question

An abstract class is a class with at least one method that is not implemented, i.e. it has no body.

```
public abstract void someMethod();  //note the semicolon at the end of the method heading
```

Since the class has at least one method that is not implemented, it must itself be declared abstract.

Here are the rules for creating an abstract class:

1. An abstract class can have private instance variables and constructors
2. An abstract class can have non-abstract methods
3. Abstract methods have no body, the heading ends in a semicolon, and must be declared abstract
4. An abstract class cannot be used to instantiate an object
5. Other classes can extend an abstract class. These classes must implement the abstract methods, or be declared abstract themselves.

Consider the following problem. A company employs two kinds of employees - hourly wage employees and salaried employees. All employees have a name and id, can change their name, receive a raise and get a pay check every week. Hourly employees have their paycheck computed by multiplying the number of hours worked by their hourly pay rate. If they have worked overtime, all hours over 40 are paid 1.5 * hourly rate. Salaried employees receive 1/52 of their salary every week.

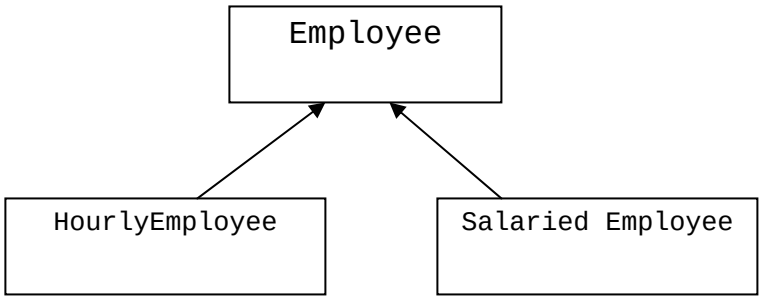
Obviously these two types of employees have a lot in common. Apply the "IS-A" relationship.

An hourly employee "**IS-A**" salaried employee?
A salaried employee "**IS-A**" hourly employee?

Neither of these is true. What about:

an hourly employee "**IS-A**" employee?
a salaried employee "**IS-A**" employee?

These relationships are true. So the hierarchy should be:



But what is a plain old employee? How do you calculate an employee's pay? You can't until you know what kind of employee you have. This problem can be solved by creating `Employee` as an abstract class. In this class, the programmer will include all of the properties that the two subclasses have in common. A constructor is created to initialize those properties. The subclasses will super to that constructor. The abstract class will **define and implement** all of the methods that both classes have in common and that can

be completed. The methods that the subclasses have in common, but cannot be completed, will be defined as abstract. The subclasses **MUST** implement these methods or be defined as abstract themselves. The completed abstract class `Employee` is shown below.

```
public abstract class Employee
{
	private String name;
	private String id;
	public Employee(String nm, String empID)
	{
		name = nm;
		id = empID;
	}

	public String getName()
	{
		return name;
	}

	public String getID()
	{
		return id;
	}

	public void changeName(String newName)
	{
		name = newName;
	}

	public String toString()
	{
		String result = "Name: " + name;
		result += "\nEmployee ID: " + id;
		return result;
	}

	public abstract void raisePay(double amount);  //don't know how to do this

	public abstract double getPayCheck(); //don't know how to do this
}
```

The methods for which the `Employee` class doesn't have enough information to complete are defined as abstract. These abstract methods must be completed by the subclasses.

The SalariedEmployee class adds one private instance variable, salary, and one accessor method, getSalary. The constructor for this class calls the Employee class constructor and then initializes the added field. The salary field is necessary to complete the getPayCheck method. The completed SalariedEmployee method is found on the next page.

```
public class SalariedEmployee extends Employee
{
	private double salary;

	public SalariedEmployee(String nm, String empID, double sal)
	{
		super(nm,empID);//always call super first
		salary = sal;
	}

	public void raisePay(double amount)		//completed method from Employee
	{
		salary += amount;
	}

	public double getPayCheck()			//completed method from Employee
	{
		return salary/52;
	}

	public double getSalary()		//new method for SalariedEmployees only
	{
		return salary;
	}

	public String toString()		//overridden toString method
	{
		return super.toString() + "\nYearly Salary: " + salary
			+ "\nCurrent Paycheck: " + getPayCheck();
	}
}
```

While you cannot instantiate an `Employee` object, `Employee` variables (references) can refer to `SalariedEmployee` and `HourlyEmployee` (when this class is created) objects.

```
Employee emp1 = new SalariedEmployee("Jill", "213", 50000);
Employee emp2 = new HourlyEmployee("Tom", "123", 22.15);
```

These references can call only the methods defined in the `Employee` class or those inherited by the `Employee` class. `emp1.getSalary()` would cause a compiler error - ***unresolved symbol*** - because the `Employee` class does not have a `getSalary` method defined.

The completed `HourlyEmployee` class is shown on the next page. Notice that two fields `hourlyRate` and `weeklyHoursWorked`, are added to the class, along with a new method, `setHoursWorked`. The fields and the method are necessary to be able to complete the `getPayCheck` method correctly. A `getHoursWorked` accessor method is also added to complete the class.

```
public class HourlyEmployee extends Employee
{
      private double hourlyRate;
      private double weeklyHoursWorked;

      public HourlyEmployee(String nm, String empID, double rate)
      {
            super(nm,empID);//always call super first
            hourlyRate = rate;
      }

      public void raisePay(double amount)          //completed method from Employee
      {
            hourlyRate += amount;
      }

      public double getPayCheck()                  //completed method from Employee
      {
            double otHours = weeklyHoursWorked - 40;

            if(otHours > 0)
               return 1.5 * hourlyRate * otHours + 40 * hourlyRate;
            else
               return hourlyRate * weeklyHoursWorked;
      }

      public void setHoursWorked(double hours)  //new method for HourlyEmployees only
      {                                         //needed to compute paycheck
            weeklyHoursWorked = hours;
      }

      public double getHourlyRate()       //new method for HourlyEmployees only
      {
            return hourlyRate;
      }

      public double getHoursWorked()      //new method for HourlyEmployees only
      {
            return hoursWorked;
      }

      public String toString()            //overridden toString method
      {
            return super.toString() + "\nPay Rate: " + hourlyRate
                   + "\nCurrent Paycheck: " + getPayCheck();
      }

}
```

Notes:

1. I assumed that the employee name was in the form **Last, First**. I am leaving this to the discretion of the teacher and student. You could create a Name class that encapsulates the first and last name. I chose not to do this.
2. My solutions are possible solutions. Your students' solutions may vary.

The next task is to create a class that will handle a company's payroll. The company has two types of employees, salaried and hourly. There is an A and AB version of this problem. The A students will most likely implement the `Payroll` class using `ArrayList` objects. The AB version requires the student to choose the best data structure to achieve the required Big-Oh performance, explain how the data is being stored, and then justify their choice.

Notes:

3. Setting the number of hours worked for each hourly employee can be done several ways. The Payroll method could iterate through the hourly employees and read the information in from a file. It could accept a data structure with employee id's and hours worked paired together. This is really up to the students and/or the teacher.
4. You might consider giving your students the `TimeCards` and `TimeCard` class.

The completed code for the A version of the `Payroll` class is found below.

```
import java.util.*;
public class Payroll
{
      private ArrayList hourlyEmp; //TreeMap for Hourly Employees - key is name
      private ArrayList salariedEmp; //TreeMap for Salaried Employees - key is name
      private ArrayList allEmp;  //HashMap for all employees - key is id

      public Payroll()
      {
            hourlyEmp = new ArrayList();
            salariedEmp = new ArrayList ();
            allEmp = new ArrayList ();
      }

      //emp has been added to hourlyEmp and allEmp
      public void addEmp(HourlyEmployee emp)
      {
            hourlyEmp.add(emp);
            allEmp.add(emp);
      }

      //emp has been added to salariedEmp and allEmp
      public void addEmp(SalariedEmployee emp)
      {
            salariedEmp.add(emp);
            allEmp.add(emp);
      }

      //private helper method - used to search the allEmp ArrayList for an
      //employee with a given id.
      private int find(String id)
      {
            for(int j = 0; j < allEmp.size(); j++)
            {
                  Employee emp = (Employee) allEmp.get(j);
                  if(emp.getID().equals(id))
                        return j;
            }
            return -1;
      }
```

```
//precondition: if an employee with id employee id
//exists, that employee is either in hourlyEmp or
//salariedEmp, not both.
public void removeEmp(String id)
{
      int index = find(id);
      if(index != -1)
      {
            Employee emp = (Employee) allEmp.remove(find(id));
            if(!hourlyEmp.remove(emp))
                salariedEmp.remove(emp);
      }
}

//precondition:  t holds all the timecards for all hourly
//employees.
//postcondition:  all hourly employees' weekly hours
//worked has been updated
public void updateWeeklyHours(TimeCards t)
{
      for(int j = 0; j < hourlyEmp.size(); j++)
      {
            HourlyEmployee emp = (HourlyEmployee) hourlyEmp.get(j);
            emp.setHoursWorked(t.getHoursWorked(emp.getID()));
      }
}

//returns the total amount of all employee's paychecks
public double getTotalPayroll()
{

      double total = 0;
      for(int j = 0; j < allEmp.size(); j++)
      {
            Employee emp = (Employee)allEmp.get(j);
            total += emp.getPayCheck();
      }
      return total;
}

public Employee getEmployee(String id)
{
      int index = find(id);
      if(index != -1)
            return (Employee) allEmp.get(index);
      else
            return null;
}

public void printHourlyEmployees()
{
      print(hourlyEmp,"Hourly");
}

public void printSalariedEmployees()
{
      print(salariedEmp, "Salaried");
}
```

```
	private void print(ArrayList list, String type)
	{
		System.out.println("=============================================");
		System.out.println("Listing of all " + type + " Employees");
		for(int j = 0; j < list.size(); j++)
		{
			System.out.println(list.get(j) + "\n");
		}
		System.out.println("=============================================\n")
	}
}
```

The TimeCard and TimeCards classes are listed below. These classes can be given to the students instead of having them write them.

```
public class TimeCard
{
	private String id;
	private double hours;

	public TimeCard(String empID, double hrsWorked)
	{
		id = empID;
		hours = hrsWorked;
	}

	public double getHours()
	{
		return hours;
	}

	public String getID()
	{
		return id;
	}
}
```

```
import java.util.*;

public class TimeCards
{
	private Map timeMap;
	public TimeCards()
	{
		timeMap = new HashMap();
	}

	public void add(TimeCard t)
	{
		timeMap.put(t.getID(), t);
	}

	public double getHoursWorked(String id)
	{
		TimeCard card = (TimeCard)timeMap.get(id);
		return card.getHours();
	}
}
```

The AB version of this problem asks students to choose data structures that will satisfy Big-Oh requirements. The specifications are repeated below.

Now that you have designed and implemented classes for the employees of a company, consider designing a class that will manage the payroll for this company. The operations for this class must include the following:

- The payroll class must be able to add hourly employees and salaried employees to the payroll.
- The payroll class must be able to delete hourly employees and salaried employees from the payroll.
- Each week, the records for each hourly employee must be updated to reflect the number of hours that were worked during that weekly pay period. **You will need to resolve how this information (weekly hours worked) is transferred to each employee object.**
- Each week, a total payroll for the company must be computed.

AB extension
If **H** is the number of hourly employees and **S** is the number of salaried employees, updating the hours worked for the hourly employees must run in at least ***O*(H log(H))** time. Computing each employee's pay check and the total payroll must run in ***O*(H + S)** time. Choose appropriate data structures to fulfill these requirements.

Add the following requirements:

- Print employee information in the following manner:
  - Print all of the hourly employees information alphabetically
  - Print all of the salaried employees information alphabetically

  This operation must be done in linear time - ***O*(H)** for hourly employees, ***O*(S)** for salaried employees
- Using an employee's id number, access to any employee's record must be done in ***O*(1)** time.
- Adding and deleting employees must be done in ***O*(log (H + S))** time. The employee's ID will be used when deleting an employee.

Explain your chosen data structures for this class. Explain how the employee data is being stored in the `Payroll` class. Justify how your data structures meet the **Big-Oh** requirements for this problem.

The `Payroll` class needs 3 data structures, one for hourly employees, one for salaried employees, and one for all employees. Since the hourly and salaried employees must be added or deleted in ***O*(log (H + S))** time and printed in ***O*(H)** or ***O*(S)** time, a `TreeMap` will satisfy the Big-Oh requirement for storing the hourly and salaried employees. Access to a given employee, given the id number, must be achieved in ***O*(1)** time. A `HashMap` will satisfy the Big-Oh requirement for storing all of the employees. Additionally, students must consider how all hourly employee records can be updated in ***O*(H log(H))** time. The `TimeCards` class uses a `HashMap` so that this requirement is satisfied. You may want to give this class to your students or let them develop their own scheme - there certainly are other ways to update the hourly employees records in ***O*(H log(H))** time or better.

This may seem very wasteful in terms of memory. It isn't. Each data structure is really storing references to the employee objects.

A solution to the AB version of the `Payroll` class is shown starting on the next page.

```
import java.util.*;

public class Payroll
{
      private Map hourlyEmp; //TreeMap for Hourly Employees - key is name
      private Map salariedEmp; //TreeMap for Salaried Employees - key is name
      private Map allEmp;  //HashMap for all employees - key is id

      public Payroll()
      {
            hourlyEmp = new TreeMap();
            salariedEmp = new TreeMap();
            allEmp = new HashMap();
      }

      //emp has been added to hourlyEmp and allEmp
      public void addEmp(HourlyEmployee emp)
      {
            hourlyEmp.put(emp.getName(), emp);
            allEmp.put(emp.getID(), emp);
      }

      //emp has been added to salariedEmp and allEmp
      public void addEmp(SalariedEmployee emp)
      {
            salariedEmp.put(emp.getName(), emp);
            allEmp.put(emp.getID(), emp);
      }

      //precondition: if an employee with id employee id
      //exists, that employee is either in hourlyEmp or
      //salariedEmp, not both.
      public void removeEmp(String id)
      {
            Employee emp = (Employee) allEmp.remove(id);
            if(emp != null)
            {
                  if(emp.equals(hourlyEmp.get(emp.getName())))
                      hourlyEmp.remove(emp.getName());
                  else
                      salariedEmp.remove(emp.getName());
            }
      }

      //precondition:  t holds all the timecards for hourly
      //employees.
      //postcondition:  all hourly employees' weekly hours
      //worked has been updated
      public void updateWeeklyHours(TimeCards t)
      {
            Set keys = hourlyEmp.keySet();
            Iterator itr = keys.iterator();
            while(itr.hasNext())
            {
                  HourlyEmployee emp = (HourlyEmployee) hourlyEmp.get(itr.next());
                  emp.setHoursWorked(t.getHoursWorked(emp.getID()));
            }
      }
```

```
	//returns the total amount of all employee's paychecks
	public double getTotalPayroll()
	{
		Set s = allEmp.keySet();
		Iterator itr = s.iterator();
		double total = 0;
		while(itr.hasNext())
		{
			Employee emp = (Employee)allEmp.get(itr.next());
			total += emp.getPayCheck();
		}
		return total;
	}

	public Employee getEmployee(String id)
	{
		return (Employee) allEmp.get(id);
	}

	public void printHourlyEmployees()
	{
		print(hourlyEmp,"Hourly");
	}

	public void printSalariedEmployees()
	{
		print(salariedEmp, "Salaried");
	}

	private void print(Map m, String type)
	{
		Iterator itr = m.keySet().iterator();
		System.out.println("===============================================");

		System.out.println("Listing of all " + type + " Employees");
		while(itr.hasNext())
		{
			System.out.println(m.get(itr.next()) + "\n");
		}
		System.out.println("===============================================\n");
	}
}
```

Explanation of the data structures:

`hourlyEmp - TreeMap`: contains references to only the `HourlyEmployee` objects

`Key` is the employee name, `value` is the employee object
Insertions and Deletions are done in ***O*(log H)** time
Traversals (alphabetical) are done in ***O*(H)** time

`salariedEmp - TreeMap`: contains references to only the `SalariedEmployee` objects

`Key` is the employee name, `value` is the employee object
Insertions and Deletions are done in ***O*(log S)** time
Traversals (alphabetical) are done in ***O*(S)** time

`allEmp - HashMap`: contains references to all of the `Employee` objects
`Key` is the employee id, `value` is the employee object
Insertions and Deletions are done in ***O*(1)** time
Traversals (alphabetical) are done in ***O*(H + S)** time
Access to an Employee object is done in ***O*(1)** time

Methods of the `Payroll` class:

`addEmp and removeEmp:` ***O*(log H)** or ***O*(log S)**
Adding/Deleting to/from `hourlyEmp` or `salariedEmp` is done in at least ***O*(log (H + S))**
Adding/Deleting to/from `allEmp` is done in ***O*(1)** time

***O*(1) + *O*(log (H))** is ***O*(log H)** / ***O*(1) + *O*(log (S))** is ***O*(log S)**: these satisfy the requirement of ***O*(log (H + S))**

`updateWeeklyHours:` ***O*(H log(H))**
This method iterates through the `hourlyEmp` map keyset, gets each employee, then accesses the id and uses the id to access the employee's time card (stored in a `HashMap`). Traversing the `keySet` of `hourlyEmp` is ***O*(H)**. Getting each employee (to get the id) from the hourlyEmp map is ***O*(log H)**. Each call to the `TimeCards` `getHoursWorked` is ***O*(1)**, since the `TimeCards` class uses a `HashMap`.

***O*(H log(H)) * *O*(1)** is ***O*(H log (H)):** this satisfies the requirement of ***O*(H log (H))**

`getTotalPayroll:` ***O*(H + S)**
This method iterates through the `allEmps` map `keySet`, gets each employee and then accesses the `getPayCheck` method to add this amount to the total. Iterating through the `keySet` is ***O*(H + S)**, each get is *O*(1) since `allEmps` is a `HashSet`. Therefore, this method runs in ***O*(H + S)** time which satisfies the requirement of ***O*(H + S)**.

`printHourlyEmployees() and printSalariedEmployees():` ***O*(H)** and ***O*(S)** respectively
Both method iterate through the `keySet` and print each employee object. Iterating through the `keySet` is ***O*(H)** and ***O*(S)** respectively. This satisfies the requirement of ***O*(H)** and ***O*(S)**.

`getEmployee:` ***O*(1)**
This method uses the employee id to lookup the employee object. `allEmps` is a `HashMap`. Lookup on a `HashMap` is ***O*(1)** which satisfies the requirement of ***O*(1)**.

Note: The keys in `allEmps` are unique `String` ids. The keys in `hourlyEmps` and `salariedEmps` are `String` names. The `String` class has a good `hashCode` method and should insure good results in a `HashMap`.