Abstract Classes and Interfaces

Abstract Methods

- A company has two kinds of employees – hourly workers who work 40 hours a week at a certain wage per hour, and salaried workers who work for an annual salary. The hourly workers get paid every week; the salaried workers once a month -- let's say every fourth week.

- I want to write a system that has a list of the company's employees; each week it runs through the list looking at each employee's data and printing a statement about how much that person should be paid.

- How do we arrange the classes to make this easy?

- Answer: Make a parent class Employee , with subclasses HourlyWorker and SalariedWorker. The staff list can be an ArrayList<Employee>

- Our payEveryone method will have a loop like this:

```
for (Employee x: staffList)
        ( <cast x into its right type).pay()
```

- If we give Employee a pay( ) method that the two subclasses override, then we don't have to cast the list variable into appropriate subclass; the runtime environment will call the subclass's method automatically.
- What body do we give the the pay() method in class Employee?

- Answer: we DON'T give it a body. This company has no generic employees, so we should never construct an element of the employee class. We make pay() an ***abstract*** method of the Employee class, which makes the class itself abstract.

  The declaration in the abstract class is

  ```
  public abstract void pay();
  ```

- An abstract class must be extended by subclasses that override its abstract methods.
- A class is abstract (and must be declared as such) if it has at least one abstract method.

- See example:
- Class Employee, SalariedWorker, HourlyWorker and StaffExample

- Advantages of abstract classes:

1. They provide a common parent class for similar but distinct classes.
2. They force the subclasses to instantiate essential methods.
3. They allow the compiler to catch things like typing errors and spelling mistakes.

Interfaces

- Here is a similar problem. I have a bunch of classes with different properties. A superclass of them does not make sense. But I still want to be able to make a list of objects of these classes and do a common operation, such as Print, to each of these objects.
- A bad solution is to take each object in the list, cast it into its native type, and run the operation on it.

- A better solution is to make an ***interface*** that contains an abstract declaration for the common method, and to force each class to ***implement*** the interface.

- Here is a simple interface declaration:

```
public interface Printable {
        void Print();
}
```