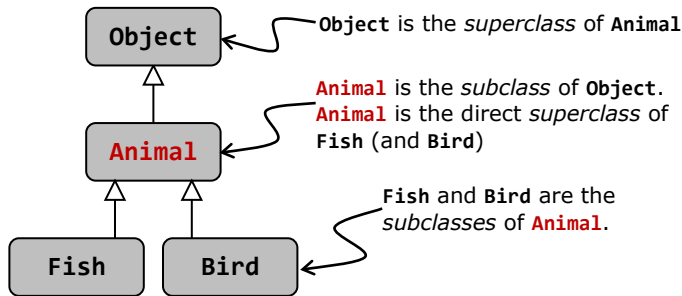
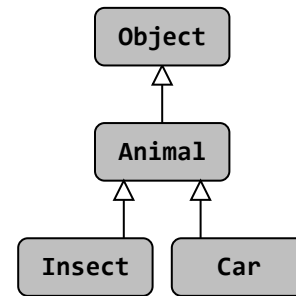


When a programmer chooses to **extend** an existing class, they do it because the existing class is a pretty good model for what they need and has many helpful features, but the class lacks certain characteristics that the programmer needs. In other words: when a programmer **extends** an existing class, they are writing a **specialization** of the existing class. In *Java* all classes ultimately extend the **Object** class. We may use a diagram to explain a class hierarchy:

This diagram shows that **Animal** extends **Object**, and both **Bird** and **Fish** extend **Animal**.



This diagram doesn't make a lot of sense since a **Car** is not an **Animal**.



When a programmer **extends** an existing class, they usually think about the three options shown below:

#### Keep existing methods:

The programmer likes an existing method in the super class and does not want to change it.

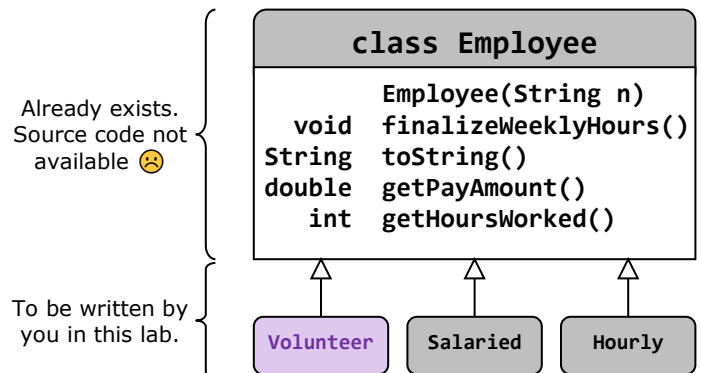
#### Modify existing methods:

The programmer sees an existing method in the super class and would like to keep the method but wants to change the behavior of that method to do its task differently.

#### Add entirely new methods:

The programmer needs to write a new method because the existing superclass methods are not appropriate.

ACME, Inc. hires you to update their payroll software that was written some years ago by a developer that has since been fired.



Code that uses the **Employee** class is shown below:

```
Employee rs = new Employee("Saujani, Reshma");
```

```
System.out.println(rs.toString()+"\n");
```

```
rs.finalizeWeeklyHours();
```

```
System.out.println(rs.toString()+"\n");
```

```
rs.finalizeWeeklyHours();
```

```
System.out.println(rs.toString()+"\n");
```

```

...Name: Saujani, Reshma
.....ID: 12888
...Hours: 0

...Name: Saujani, Reshma
.....ID: 12888
...Hours: 47

...Name: Saujani, Reshma
.....ID: 12888
...Hours: 39
  
```

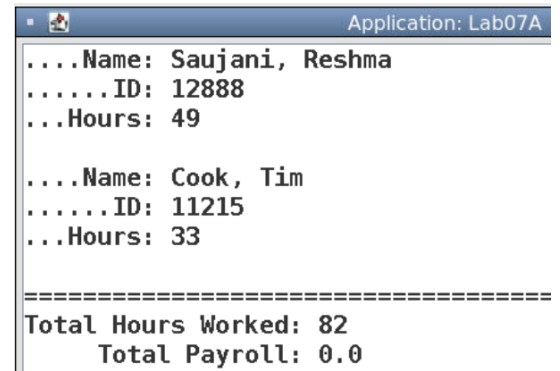
You don't have the source code to the **Employee** class (in real life: you *might* have the **Employee** source code but are not allowed to modify it because changing it might break other classes that also use **Employee**). In any event, this is how the payroll department at *ACME, Inc.* is using the **Employee** class:

```
ArrayList<Employee> empList = new ArrayList<>();
```

```
empList.add(new Employee("Saujani, Reshma"));
empList.add(new Employee("Cook, Tim"));
```

```
double totalPayroll = 0;
int totalHoursWorked = 0;
for (Employee e : empList)
{
    e.finalizeWeeklyHours();
    totalPayroll += e.getPayAmount();
    totalHoursWorked += e.getHoursWorked();
    out.append(e.toString() + "\n\n");
}

out.append("=====\n");
out.append("Total Hours Worked: " + totalHoursWorked + "\n");
out.append("      Total Payroll: " + totalPayroll + "\n\n");
```



```
Application: Lab07A
...Name: Saujani, Reshma
.....ID: 12888
...Hours: 49

...Name: Cook, Tim
.....ID: 11215
...Hours: 33

=====
Total Hours Worked: 82
      Total Payroll: 0.0
```

*ACME, Inc.* would like you to (1) create **Employee** subclasses **Volunteer**, **Hourly**, and **Salaried**, (2) fix the **getPayAmount** method in those classes since the **Employee** version always returns **0.0**, and (3) allow **Hourly** employees to receive a raise with a call to a **giveRaise(double amount)** method in the **Hourly** class.

- Volunteer: unpaid, regardless of how much they work
- Salaried: always receive the same amount of money regardless of how many hours they work
- Hourly: are paid based on their hourly rate \* hours worked

1. Run the project and make sure the output matches the screenshot shown above (the hours worked will be different each time you run the app).
2. *ACME, Inc.* would like to add volunteers to the list of employees with the following line of code:

```
empList.add(new Volunteer("Gates, Bill"));
```

Because **Volunteer** is a subclass of **Employee**, you may add it to **empList** (an array list of **Employee**). Change **Volunteer.java** so that it extends **Employee** (instead of **Object**).

If you try to run the app right now there will be compile errors because the **Volunteer** class has no constructor, and when this happens the compiler (secretly) gives you a free, no-frills constructor shown below:

Java compiler <b>incorrectly</b> tried to give you this:	This is what you really need:
<pre>public Volunteer() {     super(); }</pre>	<pre>public Volunteer(String n) {     super(n); //calls Employee constructor }</pre>
The super class, <b>Employee</b> , does <b>not</b> have a constructor with zero explicit parameters.	The super class, <b>Employee</b> , <i>does</i> have a constructor with a <b>String</b> explicit parameter.

Now add the correct constructor to **Volunteer.java**.

Uncomment line 12 in **Lab07A.java**, and re-run your app to make sure there are no other mistakes.

You should see that Bill Gates was added to the output, as shown at right. Note that the hours each employee works will be different each time you run the app.

So far there haven't been any significant changes. However, *ACME, Inc.* would like to add another line for volunteers:

```
....Name: Gates, Bill
.....ID: 10499
...Hours: 30
.....Pay: Thank you for volunteering!
```

You will now override the **toString()** method in the **Volunteer** class:

```
@Override
public String toString()
{
    String s = super.toString();
    // You add additional code to modify s
    return s;
}
```

Note that you must use **super.toString()** here. If you were to leave off the **super.**, *Java* would interpret the method call as **this.toString()** which would cause "infinite" recursion 😞.

Make sure your app now shows the correct output for Bill Gates.

Since volunteers are unpaid, returning **0.0** from the **getPayAmount()** method is appropriate. Review the employee class methods shown below.

<b>finalizeWeeklyHours()</b>	<b>toString()</b>	<b>getPayAmount()</b>	<b>getHoursWorked()</b>
Works! Do not modify in <b>any</b> subclasses	<b>Yes</b> , we override it in <b>Volunteer.java</b>	No need to override it in <b>Volunteer.java</b>	Works! Do not modify in <b>any</b> subclasses

3. Salaried employees have a yearly salary that is paid out equally every week of the 52 weeks in a year. *ACME, Inc.* would like to add salaried employees to the list of employees with this line of code:

```
empList.add(new Salaried("Bryant, Kimberly", 663_000.00)); // yearly salary is $663,000
```

Complete the **Salaried** class so that it extends **Employee**.

Decide which of the **Employee** methods need to be overridden and write code for only those methods.

When you uncomment line 13 in **Lab07A.java**, you should see the output shown at right (again, the hours worked will likely be different, but the pay should be correct).

```
....Name: Bryant, Kimberly
.....ID: 11215
...Hours: 36
.....Pay: 12750.0
```

```
=====
Total Hours Worked: 165
Total Payroll: 12750.0
```

4. Hourly employees have an hourly rate for each hour they work in a week. *ACME, Inc.* would like to add hourly employees to the list of employees with this line of code:

```
empList.add(new Hourly("Dorsey, Jack", 15.00)); // Jack's hourly rate is $15.00
```

Complete the **Hourly** class so that it extends **Employee**. Decide which of the **Employee** methods need to be overridden and write code for only those methods.

When you uncomment line 14 in `Lab07A.java`, you should see the output shown at right.

```
....Name: Dorsey, Jack
.....ID: 12888
...Hours: 44
....Rate: 15.0
.....Pay: 660.0

=====
Total Hours Worked: 194
Total Payroll: 13410.0
```

5. *ACME, Inc.* would also like to be able to give their **hourly** employees a raise by calling a `public void giveRaise(double amount)` method in the `Hourly` class. This method does not exist in the `Employee` class, so it is **not** an `@Override` method. Write the code for this method in the `Hourly` class.

To test the `giveRaise` method, add lines 16 and 17 to `Lab07A.java`:

```
14    empList.add(new Hourly("Dorsey, Jack", 15.00));
15
16    Employee jd = empList.get(4);
17    jd.giveRaise(1.25);
```

When you run the app you will get a compile error on line 17.

This is because `jd` is in an array list of `Employee`, and the `Employee` class does not have a `giveRaise` method!

```
Console  Shell
Lab07A.java:17: error: cannot find symbol
    jd.giveRaise(1.25);
    ^
symbol:   method giveRaise(double)
location: variable jd of type Employee
```

Now try this:

```
14    empList.add(new Hourly("Dorsey, Jack", 15.00));
15
16    Hourly jd = empList.get(4);
17    jd.giveRaise(1.25);
```

When you run the app you get a different compile error on line 16 because `jd` is part of an array list of `Employee`, and you may not simply change the reference type of the object being taken out of the array list.

```
Console  Shell
Lab07A.java:16: error: incompatible types: Employee cannot be
converted to Hourly
    Hourly jd = empList.get(4);
    ^
```

The way to fix this is to use a **cast** as shown below:

```
14    empList.add(new Hourly("Dorsey, Jack", 15.00));
15
16    Hourly jd = (Hourly) empList.get(4);
17    jd.giveRaise(1.25);
```

The compiler will check that the cast (`Hourly`) is indeed a subclass of `Employee`, which it is. You may now treat `jd` as an `Hourly` employee, meaning that you may call all the `Employee` methods in addition to the new `giveRaise` method located in the `Hourly` class.

Make sure your output has the updated hourly rate and a correct amount for Jack's weekly pay (similar to what is shown at right).

```
....Name: Dorsey, Jack
.....ID: 12888
...Hours: 48
....Rate: 16.25
.....Pay: 780.0

=====
Total Hours Worked: 202
Total Payroll: 13530.0
```

6. When dealing with classes and subclasses there is a **reference type** and an **actual type** that may (or may not) match exactly.

```
Employee e1 = new Employee("Rogers, David");
Employee e2 = new Hourly("Rogers, David", 9.35);
Hourly h1 = new Hourly("Rogers, David", 9.35);
Hourly h2 = new Employee("Rogers, David"); // Will not compile!!!
```

**Reference Type** (or: compile time type)      **Actual Type** (or: run time type)

The **reference type** is always known at compile time and the compiler will check that your code only calls the methods defined in that class (or a superclass). However, the **actual type** might not be known at compile time but will be known at run time it. The compiler knows that x has a **reference type** of **Employee**, but it is impossible for the compiler to know the **actual type** that x will refer to at run time.

```
Employee x;
if (Math.random() < 0.5)
    x = new Hourly("Joe", 8.0);
else
    x = new Volunteer("Sue");
```

Furthermore, the methods that will actually be selected at run time will come from the class that matches the **actual type** (or from a superclass if that method was not overridden).

The reason `Hourly h2 = new Employee("Rogers, David");` will not compile is:

The **actual type** must match (or be a subclass of) the **reference type**.

The compiler knows that the **reference type** **Hourly** has a `giveRaise` method that cannot be executed at run time since the **actual type** is **Employee**, and the **Employee** class does not have that method.

- A) Using the code above, explain why `e2.giveRaise(0.75);` will be a compile error:

**The actual type of e2 is neither \_\_\_\_\_ nor a \_\_\_\_\_ of \_\_\_\_\_.**

- B) Will `Volunteer v1 = new Employee("Smith, Jaden");` compile? Explain.

- C) Explain why `Object x = new String("GBS");` will compile.

Determine whether each of the following method calls are legal:

`x.toString()`

`x.length()`

`x.substring(0, 1);`

`x.equals("hi")`

7. When you've finished your work ask me to grade your answers to the previous step.