

Java 3 Inheritance

OOP Part 2 — Inheritance

Hierarchy
Inheritance
Overriding

Employee Superclass

A simple class around the "work" instance variable

"protected" = accessible to subclasses

"Superclass" or "Base class" -- the more general case (less specific, fewer properties)

```
/*
  The minimal Employee class.

  Demonstrate basic inheritance.

  Employees have a "work" factor which represents how much they can produce.

  -getWork() returns the productivity factor for the employee
  -totalWork() is a variant which adds in the work of any sub-employees.

  Employees do not have sub-employees, so
  their totalWork() is just their getWork().
*/

public class Employee {

    public Employee(int aWork) {
        work = aWork;
    }

    public int getWork() {
        return work;
    }

    public int totalWork() {
        return getWork();
    }

    // Instance variables
    protected int work;           // could be private (force Boss so use
    // accessors)
}
```

Boss Subclass

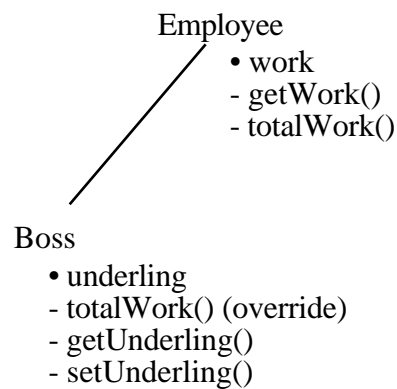
A more specialized version of Employee which has an "underling" sub-employee.
 "isa" relationship with its superclass -- has **all** the properties of its superclass + a few more properties

Overrides totalWork();

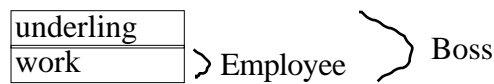
Introduces Boss specific (not present in Employee) getUnderling() and setUnderling().

Inheritance Design Diagram

The following is an excellent sort of diagram to make early in the design to think about the division of responsibility between a Superclass and its Subclass. ('•' = instance variable, '-' = method)



Memory Layout



Boss has everything that Employee has + it adds underling. Employee is the "superclass" because it is the most general-- is has the **fewest** properties.

```

// Boss.java
/*
The Boss class.
Like an Employee, but also has a single underling employee (which may be null).
The totalWork() of a Boss adds in the totalWork() of their underling if present.
Boss has getUnderling(),setUnderling() accessors which Employees do not have.
*/

public class Boss extends Employee {

    public Boss(int aWork, Employee anUnderling) {
        super(aWork); // chain to our Superclass constructor

        setUnderling(anUnderling);
        // Here, we could just set the underling directly, but it's generally
        // good form to go through the accessor in case there is some consistency
        // which needs to be maintained.
    }

    // Returns the total work due to this Boss -- their work possibly plus
    // work due to their underling. Needs to check for the case that the underling
    // is null -- sending a message to null throws an RT exception.
    public int totalWork() {
        int workFromUnderling;// this behaves like a traditional stack variable

        if (underling != null) workFromUnderling = underling.totalWork();
        else workFromUnderling = 0;

        return(getWork() + workFromUnderling);
    }

    // Standard accessors for underling

    // Returns a reference to our Employee.

    public Employee getUnderling() {
        return underling;
    }
    // NOTE The above introduces sharing -- the Boss and the caller
    // will both end up with a pointer to the same Employee.
    // If the Boss does not want the caller to change the underling state
    // there should be a comment which says so, or the Boss should
    // make and return a copy of the underling (a "clone").
    // Java does not have an idea of "const Employee" to communicate this
    // which is a shame. Sometimes in Java, such a "const" shared object will
    // have a bit set so that if anyone tries to change it,
    // an exception is thrown.

    public void setUnderling(Employee anUnderling) {
        underling = anUnderling; // note: without a garbage collector,
        // this could be a memory leak of the
        // old underling.
    }
}

```

```

// Invoke as: Boss.demo()
// Waylan is Homer's boss
// Montgomery is Waylan's boss
static void demo() {
    Employee homer;
    homer = new Employee(1);
    System.out.println("homer\t" + homer.totalWork());

    Boss waylan, montgomery;

    waylan = new Boss(2, homer);
    montgomery = new Boss(3, waylan);

    System.out.println("waylan\t" + waylan.totalWork());
    System.out.println("montgomery\t" + montgomery.totalWork());

    waylan.setUnderling(null);
    System.out.println("montgomery\t" + montgomery.totalWork());

    /*
    output...
    homer 1
    waylan 3
    montgomery 6
    montgomery 5
    */
}

private Employee underling = null;
/*
New objects in the heap have all their instance vars set to 0.
Depending on the type, 0 has the value:0, 0.0, false, "", or null.
Therefore, the above doesn't really do anything.
This also demonstrates that it is possible to assign a default
value to an instance variable right where it is declared --
this can create a little uncertainty since it's easy to expect
that the instance vars have their values set in a constructor.
*/
}

/*
Questions:
- in totalWork: Why call getWork() -- why not just access work directly ?
- in totalWork: Why call underling.totalWork() instead of underling.work() ?
- Of what classes may the underling object be?
- Why did Employee need to provide a trivial definition of totalWork() ?
*/

```

Inheritance Code Notes

No matter what code is being executed, the receiver remains constant (even if it is of a different class) and the receiver never forgets its class.

Send yourself a message rather than manipulate the instance variable — important in the frequent case that accessors are maintaining some internal consistency. (e.g.

```
RA.changeMood()
```

Subclassing "isa" -- Boss isa Employee since it has all the properties which Employees have (+ the underlying properties specific to Bosses)

OOP Substitution Rules

What does "Employee x;" really say about x at CT in the source code? "x is an instance of the Employee class?" **NO**

"x is **at least** an Employee" -- i.e. x has all the properties that Employees have. It may be a subclass of Employee, in which case it will have additional properties.

Rule: a Subclass is a valid stand-in for its Superclass. e.g. a Boss can go anywhere an Employee can go (since it has all the necessary properties). The reverse is not true: an Employee cannot go anywhere a Boss can go.

```
Employee e = new Employee(1);
Boss b = new Boss(1, null);
...
e = b; // YES, e points to something with all of the props of an Employee
      // i.e. e.getWork() must work.

b = e; // NO, b requires something with all the Boss props, and e is insufficient
      // i.e. b.getUnderling() must work (it could not if b referred to an
      // Employee but not a Boss.)
```

Example Where you need substitution

You would like to be able to pass both Employee objects and Boss objects to the employeeFeedback() method -- but what type can you make the argument?

```
// This function demonstrates a useful case of using a variable
// with type Employee in the source to refer flexibly to either
// an Employee or Boss object at run-time.
// Passing a Boss to this method works fine...
//   Boss b = new Boss(..);
//   employeeFeedback(b);
// A subclass is a valid stand-in in contexts needing the superclass.
private static void employeeFeedback(Employee employee) {
    if (employee.totalWork() <= 1.0)
        System.out.println("Back to work Slacker!");
    else
        System.out.println("Keep up the good work.");
}
```

Objects Always Remember Their Class

At RT, objects always remember their true class.

The (e.totalWork()) call above correctly resolves to either the Employee or Boss totalWork() method depending on the RT class of the passed in object.

The CT type in the source code is fixed at Employee which we know means "at least an Employee"

Watch Out For Sharing

getUnderling() returns an Employee. But all object references are shallow, so the caller and the receiver will end up both owning pointers to the same Employee. The code for both objects will need to account for this sharing.

- 1) If the shared object is immutable (cannot be changed), then there's no problem (this is the approach String uses).
- 2) The Boss class should, in its comments, state how it wishes the client to behave with the shared object.
- 3) Some objects will return a reference in such a way that trying to change it throws a RT exception -- this is a way to communicate that they do not wish it to be changed (cheezy but effective IMHO).
- 4) If the receiver really doesn't want there to be any danger, then they should use an immutable object, or they should make a deep copy and return that to the caller (and the comments could communicate that this is what is happening to avoid in confusion for the caller).
- 5) It would be nice if Java had a "const" keyword to at least suggest the authors intention, even if the RT exceptions are used as backup.