Java Inheritance

Handout written by Julie Zelenski.

Inheritance is a language property specific to the object-oriented paradigm. Inheritance provides a unique form of code-sharing by allowing you to take the implementation of any given class and build a new class based on that implementation. This new subclass starts by inheriting all of the data and operations defined in the superclass. It can then extend the behavior by adding additional data and new methods to operate on it. The subclass can also extend or replace behavior in the superclass by overriding methods that were already implemented. The inheritance hierarchy can be many levels deep. At the root of the entire class hierarchy is the `Object` class, from which all classes eventually derive in Java.

The `Employee` class

The trivial `Employee` class tracks the hours worked by an employee along with their hourly wage and their "attitude" which gives you a rough measure of what percentage of time they are actually productive:

```
/*
 * Class: Employee
 * ---------------
 * Employees have a "productivity" factor that represents how much
 * they produce -- accessed by getProductivity(). getTeamProductivity() is
 * a variant which adds in the work of any underlings.  Employees do not
 * have underling, so their getTeamProductivity() is just getProductivity().
 */

public class Employee {

   public Employee(int wage, int hours, double att)   // constructor
   {
      wagePerHour = wage;
      numHoursPerWeek = hours;
      attitude = att;
   }

   public double getProductivity()
   {
      return numHoursPerWeek*attitude;
   }

   public double getTeamProductivity()
   {
      return getProductivity();
   }
```

```
   public int askSalary()
   {
      return wagePerHour*numHoursPerWeek;
   }
                                          // instance variables
   protected double attitude;
   protected int numHoursPerWeek, wagePerHour;
   // these could be private (would force subclasses to use accessors)
}
```

Subclassing `Employee`

"Subclassing" and "superclass" are the broadly accepted terms in the OOP world. The C++ subculture sometimes calls a superclass a "base" class and a subclass a "derived" class. Derive and subclass can both be used as verbs, but not in polite conversation. e.g. "So then thinking quickly, I just derived off of the window base class and I was golden."

The following creates a subclass of `Employee`, the `Boss` class. The `Boss` is a more specialized version of `Employee` that adds a vector of "underling" sub-employees. Note that Boss represents an *isa* relationship with `Employee`. A `Boss` is a particular type of `Employee`, so it has all the same properties plus a few more. `Boss` overrides the team productivity method to add in the work done by the underlings and adds a few new methods of its own dealing with underlings. In addition, bosses have the annoying habit of lying about their salary, they always say they make twice as much as they really do.

```
/*
 * Class: Boss
 * -----------
 * Like an Employee, but also has a Vector of underling employees (which
 * may be empty). The getTeamProductivity() of a Boss adds in the
 * team productivity of any underlings. Also they lie when asked about
 * their salary.
 */

public class Boss extends Employee { // subclass of Employee

   public Boss(int wage, int hours, double att, Employee underling)
   {
      super(wage, hours, att);      // chain to our superclass constructor

      underlings = new Vector();    // new empty vector of underlings
      if (underling != null)
         addUnderling(underling);

      // could add underling to vector directly, but is considered
      // good form to use our own accessor in case any constraints
      // need to be adhered to when changing the state
   }

   public double getTeamProductivity()
   {
      double sum = getProductivity();     // start with own productivity

      for (int i = 0; i < underlings.size(); i++) // add underling contrib
```

```
        sum += ((Employee)underlings.elementAt(i)).getTeamProductivity();

      return sum;
   }


   public int askSalary()
   {
      return 2 * super.askSalary(); // inflate the inherited version
   }


   public void addUnderling(Employee anUnderling)
   {
      underlings.addElement(anUnderling);
   }

   public void removeUnderling(Employee anUnderling)
   {
      underlings.removeElement(anUnderling);
   }

   public boolean isDirectReport(Employee anUnderling)
   {
      return underlings.contains(anUnderling);
   }

   protected Vector underlings;     // boss adds one instance variable
}
```

- In Employee's `getTeamProductivity` method, why call `getProductivity`—why not just do the calculation using the ivars directly?
- Why does Boss call `getTeamProductivity` on each underling instead of `getProductivity`?
- Why do we have to cast the result from Vector `elementAt` to `Employee` when in tabulating the underling productivity? What will happen if the object is really a `Boss`?
- Why did the `Employee` class need to provide a trivial definition of `getTeamProductivity`?
- Why call the super version in Boss's `askSalary` instead of copying code?
- Why does `Boss` have several methods to allow changing and searching the underlings list? Why not just have an accessor that gives out the Vector directly and clients can do with it what they want?

Super

To access the inherited version of a method, we use `super` as the receiver when sending a message. This sends a message to the receiving object, but starts the method lookup at the parent class.

It searches for the first class above it in its inheritance chain that implements the method and uses that version. When a class overrides a method to tack something extra onto the usual processing it calls the inherited version using super.

Unlike C++, the Java language does not have a construct that allows you to jump over the parent version of a method and get to the version defined in a specific grandparent or other ancestor. Using `super` always starts looking from the parent class and stops at the first one found.

In almost all cases, `super` is used within the definition a method to include the parent version of the method being defined, e.g. a subclass overrides to replace the `binky` method, but makes a call to `super.binky` within the definition to include the parent processing along with the subclass's extensions. However, you can actually use super to call the parent version of any method, not just the one you are currently defining. For example, the subclass's `binky` method could also make a call to `super.winky` if desired. (This is rarely needed, but mentioned here for completeness.) When calling the parent version of a constructor, you use keyword `super` by itself, with no method name following (see `Boss` constructor for an example). The syntax for constructors is just a bit of oddity overall.

Inheritance and constructors

One small wrinkle on what the subclass inherits concerns constructors. Constructors are considered to be tightly coupled to the class they are defined in, and thus are not inherited. For example, attempting to use the 3-argument `Employee` constructor to build a `Boss` would not work:

```
b = new Boss(25, 40, .50); // NO! Boss has no 3-arg constructor!
```

The only constructor actually defined in `Boss` is one that takes four arguments, where an underling is the additional argument. If we wished `Boss` to also respond to a 3-argument constructor like that of `Employee`, we would add this constructor to the `Boss` class and then call `super` to use the inherited version from the `Employee` class.

```
public Boss(int wage, int hours, double att)
{
   super(wage, hours, att);
}
```

This looks a little silly since defining an override of the method and then just calling `super` is usually pointless, but for constructors it is necessary. Every constructor you wish to have for a class must be defined in that class itself, it will not inherit any additional constructors. In order to access any of your parent constructors, you have to define a trivial override that just chains to the superclass constructor.

Remember that if we define absolutely no constructors in a class, then the compiler supplies the default constructor that takes no arguments and clears the object's instance variables. That constructor will attempt to call the parent's zero-argument default constructor to initialize the parent portion of the object. If no such constructor exists, you will get a compile-time error.

Class Compatibility

An instance of a class can "take the place" of an instance of its superclass. A class always has all the properties of its superclass, and potentially more. Any message you could send to an `Employee` will definitely be understood by a `Boss`. Thus, it is always okay to substitute a `Boss` object for an `Employee` object. For example, consider the method `addUnderling` which takes one argument of type `Employee`, would it be okay to pass a `Boss` instead? Sure! What can you guarantee about the type of the object passed to this method? The object will be at least an instance of the declared class, which is to say it can be an `Employee` or any subclass of `Employee`, like `Boss`. In the following fragment, note we can freely assign a `Boss` into an `Employee` variable:

```
Employee emp;
Boss boss;

boss = new Boss(30, 40, .60, null);

emp = boss; // OK. Boss has all the properties of Employee so boss can
            // stand in for an Employee
```

Assigning a subclass into a variable of the superclass type is called *upcasting*, since you are moving upwards on the inheritance tree. Upcasting is always safe and legal.

The reverse is not true. In general, a subclass has added methods and fields that are not present in the superclass and thus it is not okay to substitute an instance of the superclass in for the subclass since it will not have those extra features. For example, a `Boss` responds to methods such as `addUnderling` that are not understood by an `Employee` object. If we try use an `Employee` object where a `Boss` is expected, it will not have all the right features. Trying to substitute an `Employee` for a `Boss` fails to compile in Java, as in this example:

```
Employee emp;
Boss boss;

emp = new Employee(15, 40, .75);

boss = emp; // doesn't compile. boss type requires all Boss props and emp
            // is insufficient. What if we send addUnderling message?
```

The compiles disallows this because any random `Employee` does not necessary have all the properties required of a `Boss` object. For example, what happens if we try to send that `Boss` the `addUnderling` message?

Sometimes, though, the `Employee` object might really be a `Boss` (e.g. it was upcast in an earlier operation) and we may really want to force such an assignment through. Assigning a superclass into a variable of the subclass type is called *downcasting*, since you are moving down the inheritance tree. Downcasting is not safe or legal in all cases, so Java requires you to take a extra step to verify the class compatibility at runtime.

We do this by using a typecast to cast from one class to another. This “class cast” looks like the standard C typecast, but it is quite different from the compile-time "I know best" cast of C. It is a dynamic operation that at runtime determines whether the particular `Employee` is truly compatible with `Boss` (remember all objects know what type they are) and then allows the cast and assignment to go through if okay or raises an exception.

```
Employee emp1, emp2;
Boss boss;

emp1 = new Employee(15, 40, .75);
emp2 = new Boss(15, 40, .75, null);

boss = (Boss)emp1;   // fails at RT: class cast exception
boss = (Boss)emp2;   // downcast succeeds, emp2 is compatible w/ Boss
```

You use the downcast often in situations such as retrieving objects from a `Hashtable` or `Vector`. Since these collection classes are designed to work for all types, they use `Object` as the declared type for passing and returning elements. If you want to restore the upcasted `Object` to somewhere lower on the hierarchy, you must use the safe runtime class cast. This is not entirely unlike the step where we hat to cast the `void *` returned from `ArrayNth` to what we expected it to really be. The big difference is that C’s typecast is completely unsafe with no runtime check for correctness (there is no RT type information to even check against if we wanted to), unlike Java’s safe and controlled class cast that verifies the compatibility of the cast at runtime.

Compile-Time Message Type-checking

In Java, you cannot just send any random message to any given object. When compiling a message send, the compiler makes sure that the receiving object is capable of responding to such a message. The compiler uses the declared type of the receiver and then verifies that that class has a defined or inherited `public` method matching the name and signature of the call. If none is found, a fatal error is triggered.

Note this is a compile-time process! The compiler operates using only the compile-time declared type, without regard for what the actual variant might be at runtime. For example, if you assign a `Boss` object into a `Employee` variable, as far as the compiler is concerned, that object is now just an `Employee` and is treated accordingly. For example:

```
Employee emp;

emp = new Boss(30, 40, .60, null);
```

```
emp.addUnderling(other);   // not okay! Employee doesn't have addUnderling!
```

An object never forgets its class and never morphs from one class to another during its lifetime. The above assignment statement does not change the `Boss` into an `Employee`. The object is still a `Boss` and always will be. For example, if you were to send this object the `askSalary` message, it would use the `Boss` version. However, by storing the `Boss` into a variable of type `Employee`, we are "losing" a bit of information in the compiler's eyes. The compiler will not allow us to send this object Boss-specific messages (such as `addUnderling`), because in the CT world, it only knows that the object is at least an `Employee`, but nothing more can be assumed. The compile-time message type-checking will only allow us to send `Employee` or above messages to this object.

Run-Time Message Resolution

When the compiler encounters a statement such as `obj.askSalary` it knows it will be sending the `askSalary` message to the object, and after checking that `obj`'s class has such a method, the compiler is happy. Given there may be more than one class that responds to the `askSalary` message, the compiler cannot tell for sure what the actual code is that will be used. However, at runtime, there will be no ambiguity. An instance of a class never forgets what class it belongs to. When dispatching the message at runtime, the known type of the receiving object determines exactly which version of the `askSalary` method is invoked. This is run-time polymorphism where the right thing just happens.

Note that it does not rely on the compile-time type to determine which method to invoke. For example, consider the case of a `Boss` who happens to have another `Boss` as his underling. The underling is typed as an `Employee` object, but at runtime what if it were a `Boss` object instead? When we send the `getTeamProductivity` method to our underling, we will invoke the `getTeamProductivity` method of the `Boss` class, not the `Employee` class, because the object itself knows it is a `Boss` object and thus uses its own version of the method.

Run-time Polymorphism

Why is it that we might want to "forget" what class an object is? You might figure that you would always want to give the precise type (`Employee`, `Boss`, `Intern`, whatever) when declaring each variable rather that only identifying them as a generic `Employee`. However, many times, it is extremely useful to treat all `Employees` similarly without regard for their specific subtype. For example, we may want to manage all the employees in a company and we don't want to separate the `Bosses` or `Interns` from everyone else. It is more convenient to create the full list of all `Employees` and then go through and send messages to get their productivity or raise their salaries. For different `Employee` sub-types, these methods may be implemented differently. It is the beauty of polymorphism that each subclass can have its own distinct way of responding to the same message and yet the client doesn't need to know or plan for this, it treats all objects as mere `Employees`.

Polymorphism Example

Consider the following simple example to demonstrate the power of polymorphism. There are three types of shapes subclassed from the generic `Shape`. Each specialization of shape responds to the `draw` message differently.

```
public abstract class Shape {
   abstract public void draw();     // abstract means no implementation here
}

public class Circle extends Shape {

   public void draw()
   { ... }
}

public class Rectangle extends Shape {

   public void draw()
   { ... }
}

public class Polygon extends Shape {

   public void draw()
   { ... }
}
```

Now, here's a sample use of an array of various Shapes:

```
{
   int i;
   Shape[] shapes;      // array of random shapes

         ...            // create array and add new shapes

   for (i = 0; i < shapes.length; i++)
      shapes[i].draw();       // polymorphism at runtime
}
```

The shapes array contains a bunch of objects, all of which are subclasses of the `Shape` class. At compile-time, there is no way to know which shapes will be `Circle`, which will be `Rectangle`, and which will be `Polygon`. The only way for the line `shapes[i].draw()` to do the right thing at run-time is for each object to know what class it is at run-time and so direct the `draw` message to the method for its class. This is canonical runtime-polymorphism behavior and makes the object itself responsible for matching the message with the right method rather than the programmer. The client doesn't need to care that some shapes are `Circle`s, some are `Square`s, and so it. It is incredibly convenient for the client to not bother to track and just "forget" which shapes are of which class, knowing that the object itself is taking the responsibility to "do the right thing" on its own. Wow!