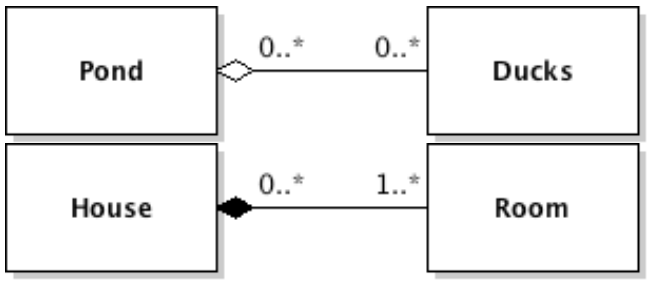
CECS 277 – Lecture 2 – Inheritance

The purpose of writing classes is to create a library of self contained (encapsulated) objects that can be later be used or reused whenever a particular object is needed. So far, when writing our programs, we have been using a concepts known as aggregation and composition, the idea that class A "has a" class B. Aggregation is the instance when if when class A is destroyed, the class Bs that it contained continue existing. (Example: A pond contains 0 or more ducks. The ducks may continue to exist elsewhere if the pond no longer existed). Composition is the opposite, if class A is destroyed, then all of the class Bs are destroyed as well. (Example: A house contains 1 or more rooms. If the house is destroyed, then all of the rooms are destroyed as well).

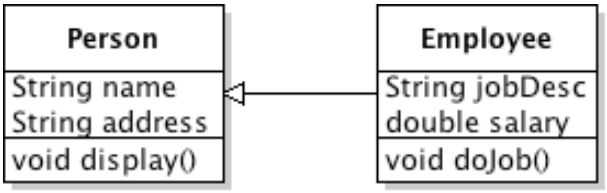
UML Class Hierarchies – aggregation and composition



Inheritance –

Inheritance is the idea that you can define a new class based on an existing class. Anything that has a "is a" relationship could be inherited. This allows variables and methods from the existing class to automatically become part of the new class. This cuts down on a lot of duplicated code. The new class simply uses the keyword extends and names the class to be inherited from.

UML Class Diagram – Employee inherits variables and functionality from Person.



The inheritance is one directional though, the Employee "is a" Person, but a Person may not be an Employee. The Person class can be extended to other types of classes, such as Customers, or Contacts. The Employee class can be extended to create different types of employees, such as Secretaries and Engineers. All of these subclasses can be easily written since most of the functionality was initially created in the superclass. This frees the programmer from having to write and test similar functions for each of the classes. Plus, if any modifications need to be made, it can be done in just the single location, rather than trying to figure out where else you might have used that function.

Access Modifiers –

When you extend a class you inherit any variables or methods that are not private or final. If you create your instance variables as private, you will need to use getter and setter methods, defined in the superclass, to access them. There are other possible access modifiers that you may use, but they have nearly the same vulnerabilities as any public variable would.

Modifier	Class	Package	Subclass	Anywhere
public	yes	yes	yes	yes
protected	yes	yes	yes	no
default	yes	yes	no	no
private	yes	no	no	no

Overriding –

If you create any variables or methods in a subclass that have the same name as in the superclass, then the version in the subclass overrides the version in the superclass, basically taking precedence and effectively hiding the superclass's version.

This allows a programmer to implement greater functionality in particular subclasses. If there are several subclasses that all extend from a superclass and you want most of them to have a default behavior, then they would all use the method from the superclass. The others would override the function with the new method by the same name.

Super –

If you override a method but you still want to perform the same actions that the superclass's method had, then you can use the keyword super. It can be used in a similar way as an object's variable name, by using the dot operator to call the desired method, or by itself to call the constructor. The call to the superclass can then be followed by other statements for your subclass.

```
super.doTask(); //superclass method call

super();  //default superclass constructor
```

Polymorphic Types –

When you are creating instances of your classes, you can make objects from either the superclass or the subclass depending on the type of functionality you want for your object. There may be some cases in which you will need to create an instance of your subclass but as an object of the superclass. Ex. An array of People, some of which are Employees. In most cases, this is acceptable, as the subclass data is not lost, but some of the functionality may be lost due to it not being an object from the subclass. Most overridden methods will still use the subclass version of the method though

```java
/**
 *  Person is a simple representation of a person
 *  @author Shannon Foss
 *  @version Date: Jan 25, 2012
 */
public class Person {
	/** Person's name */
	private String name;
	/** Person's address */
	private String address;

	/** Initializes a person using string parameters.
	 *	@param n  The name.
	 *	@param a  The address.
	 */
	public Person(String n, String a){
		name = n;
		address = a;
	}
	/** Displays a person */
	public void display(){
		System.out.println(name);
		System.out.println(address);
	}
	/** Accessor for a person's name
	 * @return name
	 */
	public String getName(){
		return name;
	}
	/** Accessor for a person's address
	 * @return address
	 */
	public String getAddress(){
		return address;
	}
	/** Mutator for a person's address
	 * @param a The address.
	 */
	public void setAddress(String a){
		address = a;
	}
}
```

```java
/**
 *  Employee is a representation of an employee extended Person
 *  @author Shannon Foss
 *  @version Date: Jan 25, 2012
 */
public class Employee extends Person{
     /** Person's name */
     private String jobDesc;
     /** Person's name */
     private double salary;
     /** Initializes an Employee using input parameters.
      *    @param n  The name.
      *    @param a  The address.
      *    @param d  The job description.
      *    @param s  The salary.
      */
     public Employee(String n, String a, String d, double s) {
          super(n, a);
          jobDesc = d;
          salary = s;
     }
     /** Method to perform job */
     public void doJob(){
          System.out.println("My name is "+ getName());
          System.out.println("Work  Work  Work");
     }
     /** Displays an Employee */
     public void display(){
          super.display();
          System.out.println(jobDesc);
          System.out.println(salary);
     }
     /** Accessor for an employee's job description
      * @return jobDesc
      */
     public String getJobDesc(){
          return jobDesc;
     }
     /** Accessor for an employee's salary
      * @return salary
      */
     public double getSalary(){
          return salary;
     }
```

```
	/** Mutator for an employee's job description
	 * @param j The job description.
	 */
	public void setjobDesc(String j){
		jobDesc = j;
	}
	/** Mutator for an employee's salary
	 * @param s The salary.
	 */
	public void setSalary(double d){
		salary = d;
	}
}

/*	Author: Shannon Foss
	Date: Jan 24, 2012
	Program: TestEmployee.java - Tester for Employee class.
	Desc: Example of writing an extended class. */
public class TestEmployee {
	public static void main(String[] args) {
		Employee e1 = new Employee("Paul","123 Fake St",
		"Desktop Support",35000);
		Person p1 = new Person ("John","456 Main St");
		//does not work - a person is not an employee
		//Employee e2 = new Person("George","789 Elm St");
		//but an employee is a person
		Person p2 = new Employee("Ringo","100 Oak St",
		"Network Admin",75000);
		e1.display();
		e1.doJob();
		p1.display();
		p2.display(); //uses employee display but not doJob()
	}
}
/*Output:
Paul                                    Ringo
123 Fake St                             100 Oak St
Desktop Support                         Network Admin
35000.0                                 75000.0
My name is Paul                         */
Work  Work  Work

John
456 Main St
```