CECS 277 – Lecture 2 – Class Relationships

When creating programs with several classes those classes will often have relationships between each another. There are several different types of relationships: dependency, aggregation, composition, and inheritance, they each represent a different idea of how two classes relate to each other.

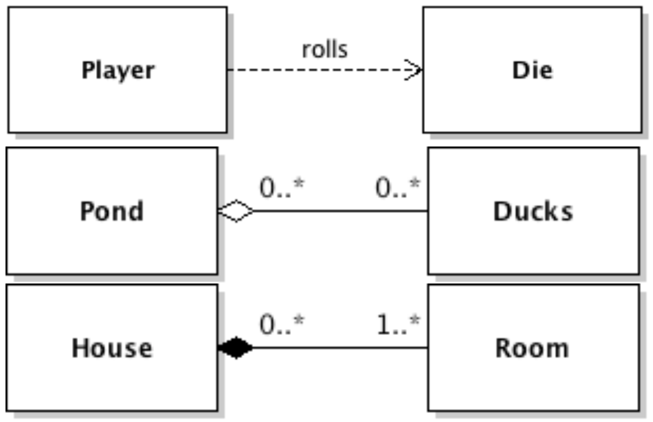
Dependency – is the case where class A 'uses' an instance of class B. An object of type B might be passed in to one of class A's functions in order to use one of B's functions. It is often beneficial to minimize dependencies in your code. Whenever B is modified, then A will usually need to be changed too.

Aggregation – is the case where class A 'has' one or more references to class B. An instance of class B is created elsewhere and then passed in and the reference is set. Then, if class A is ever destroyed, any objects of type B that class A referenced will continue to exist wherever they were created.

Composition – is a stronger type of aggregation, class A 'owns' an instance of class B. Class A constructs one or more instances of class B, so if A is ever destroyed, so are any instances of B that A owned.

Examples:

UML Class Hierarchies – Dependency, Aggregation, and Composition

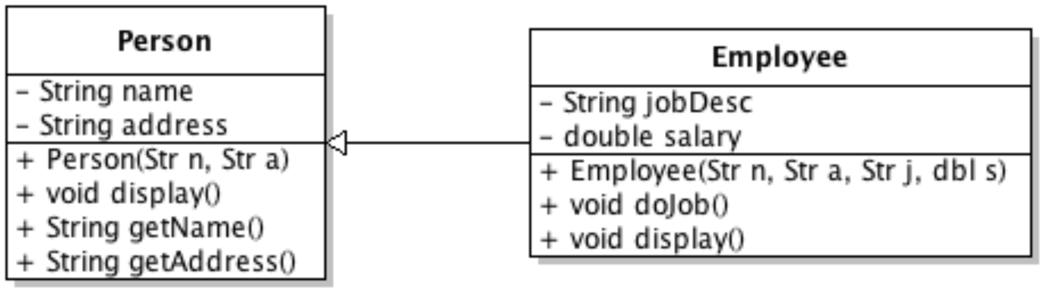


Dependency	Aggregation	Composition

```
public class Die {
  public int roll(){...}
}
public class Player{
  public int turn(Die d)
    d.roll();
  }
}
```
	```
	public class Duck {...}
	public class Pond {
	  ArrayList<Duck> ducks;
	  public void add(Duck d){
	    ducks.add(d);
	  }
	}
	```
		```
		public class Room {...}
		public class House {
		 Room [] r;
		 public House( ){
		  r = new Room[5];
		  for(int i=0;i<5;i++){
		    r[i] = new Room();
		  }
		 }
		}
		```

Inheritance – is defining a new class from an existing class. Anything that has an 'is a' relationship would be inheritance (a Dog is an Animal). This allows variables and methods from the existing class to become a part of the new extended class. This cuts down on a lot of duplicated code. The new class (the subclass) uses the keyword `extends` to identify the class it is inheriting from (the superclass).

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



The inheritance is one directional though, the Employee 'is a' Person, but another Person object might not be an Employee. The Person class can be extended into other types of classes, such as Contacts, or Customers. The Employee class can then be extended into different types of employees, such as Secretaries, Accountants, or 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 superclass, rather than trying to figure out where else you might have used that function.

Access Modifiers – When you extend a class you inherit all variables and methods, but you do not have direct access to anything that is private. If you make your instance variables private, you will need accessor methods in the superclass in order to use them. You can alternatively make your data members protected, this will allow direct access for any subclasses (even sub-subclasses), but all other classes will still be restricted. You should always avoid making your data members public. This could allow for unwanted modifications of your variables and lead to invalid values. Use accessor/mutator methods to check any modifications being made so that you can avoid any invalid values.

Modifier	Class	Subclass	Anywhere
public	yes	yes	yes
protected	yes	yes	no
private	yes	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 will override the version in the superclass, basically taking precedence and effectively hiding the superclass's version. This allows a programmer to implement refined functionality in particular subclasses. If there are a few subclasses that should do something different than the default version in the superclass, then those subclasses should override the method by using the same method signature as the original method in the superclass. Then, when the method is called from objects created from those subclasses, they will use the overridden version.

Super – If you need to execute the superclass's version of an overridden method, then you can call it using the keyword `super`. If you are calling super from a constructor, it must be the first statement executed.

```
public Subclass(){
      super();          //calls superclass constructor
      //set other values
}
public void display() {
      System.out.println("Subclass display function");
      super.display(); //calls superclass display method
      //display other information
}
```

```
/**
 *  Person is a simple representation of a person
 *  @author Shannon Cleary
 */
public class Person {
      /** First name of the person */
      private String name;
      /** Location where the person lives */
      private String address;

      /** Initializes a person's name and address
       *  @param n     First name.
       *  @param a     Street address.
       */
      public Person( String n, String a ) {
            name = n;
            address = a;
      }
      /** Displays a person's information */
      public void display() {
            System.out.println( name );
            System.out.println( address );
      }
      /** Accesses a person's name
       * @return the name of the person object.
       */
      public String getName() {
            return name;
      }
      /** Accesses a person's address
       * @return the street address of the person object
       */
      public String getAddress() {
            return address;
      }
}
```

```java
/**
 *  Employee is a representation of an employee that extends Person
 *  @author Shannon Cleary
 */
public class Employee extends Person {
	/** The Employee’s job description */
	private String jobDesc;
	/** The Employee’s annual income */
	private double salary;

	/** Initializes an Employee.
	 *  @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 have an employee do their job */
	public void doJob() {
		//cannot access name directly, must use getName()
		System.out.println("My name is "+ getName());
		System.out.println("Work  Work  Work");
	}

	/** Displays an Employee */
	@Override
	public void display() {
		super.display();
		System.out.println(jobDesc);
		System.out.println(salary);
	}
}
```

```
/**
 *  Tester for Employee and Person classes
 *  @author Shannon Cleary
 */
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" );

            //this 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 is an Employee, can access any members in employee
            e1.display();
            e1.doJob();
            //e1 is a Person, can access any public members in Person
            System.out.println( e1.getName() );

            p1.display(); //a person uses the Person class display()

            //Employee instance in a Person object
            p2.display(); //uses Employee display() since it overrides
            //p2.doJob(); //<-- error, does not exist for Person class
      }
}
/*Output:

Paul
123 Fake St
Desktop Support
35000.0
My name is Paul
Work  Work  Work
Paul

John
456 Main St

Ringo
100 Oak St
Network Admin
75000.0
*/
```