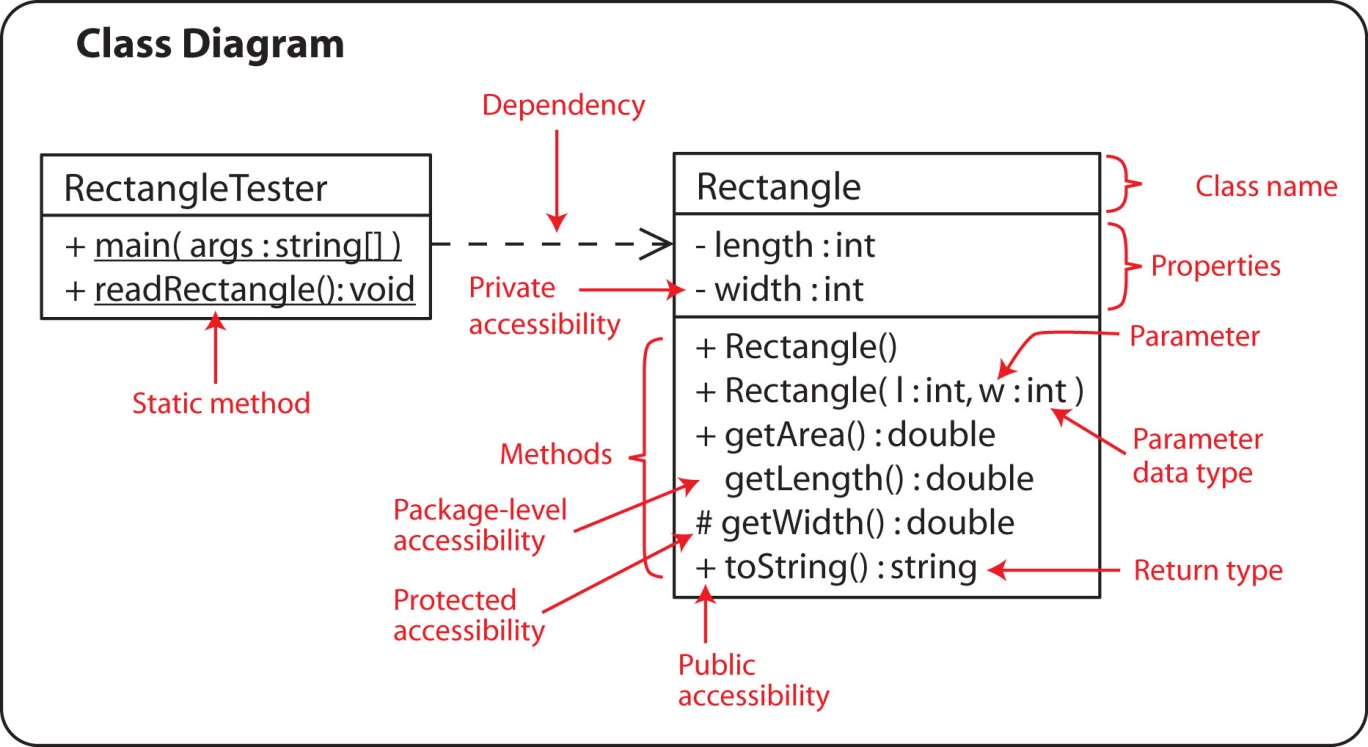
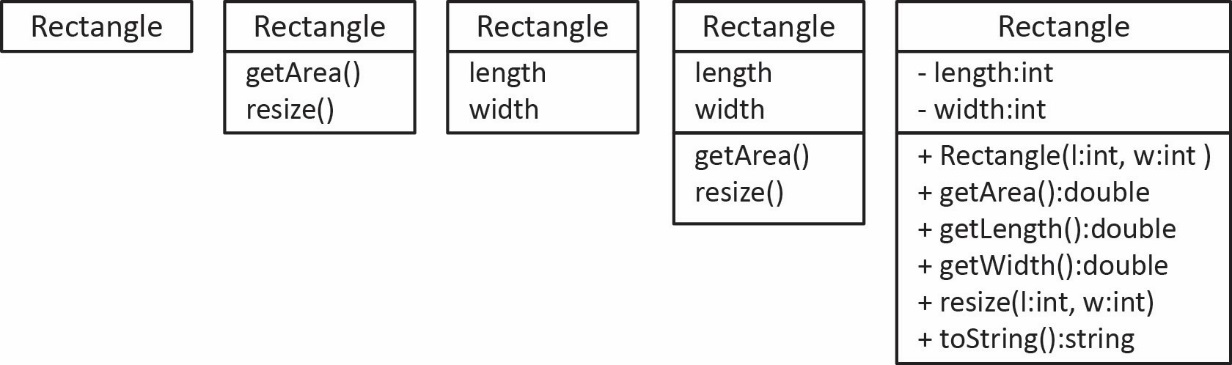
**Class, Object, & Package Diagrams**

**Class and Object Diagrams**

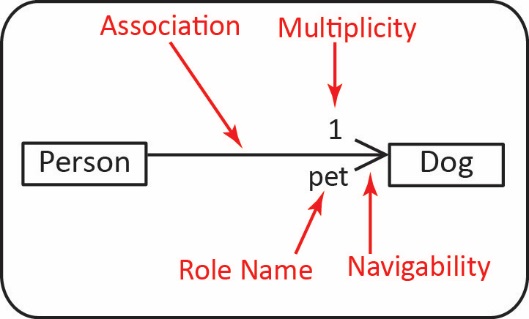
1. **Class Diagram** – Used to show the static structure of a system. It depicts the classes and their relationships. The items below should be clear except perhaps the the *dependency* relationship which we consider later. Note: static methods are underlined.



1. We can show a class in various degrees of detail as shown below, depending on the stage of modeling.



1. **Association, Role Name, Navigability, and Multiplicity**
2. Example:



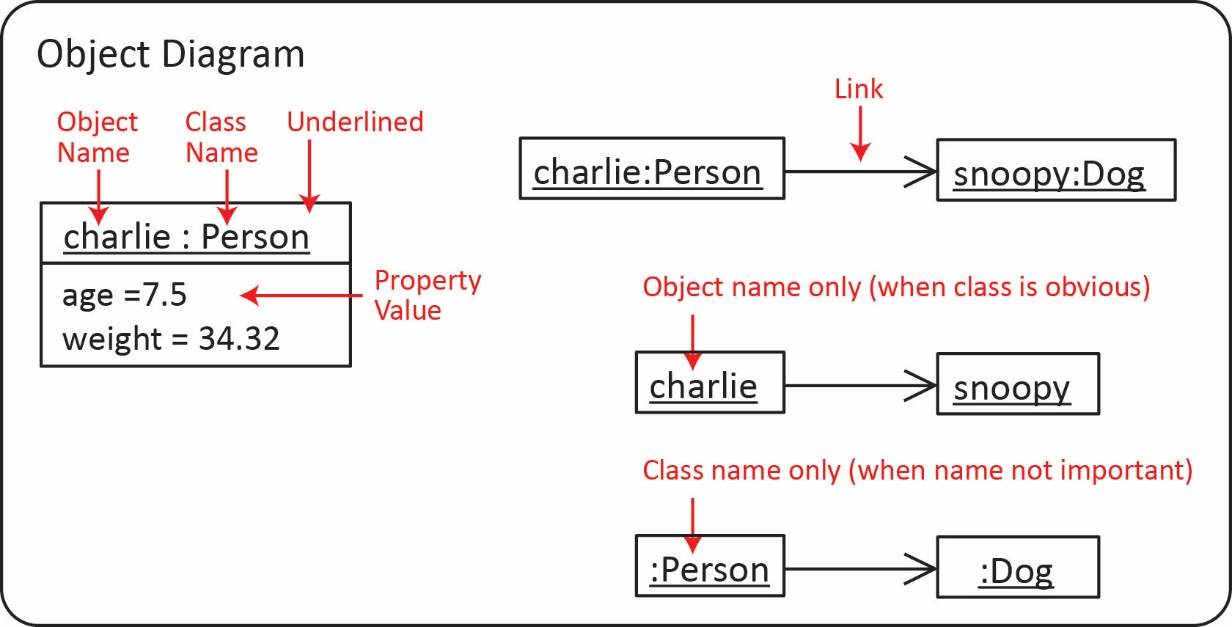
1. Note the following:

* The *association* is read: “A *Person* has-a *Dog*.”
* The *multiplicity* indicates that a *Person* has exactly one *Dog*.
* The *navigability* indicates that a *Person* knows who her *pet* is but a *Dog* does not know who its owner is.
* The *role name* indicates that the *Person* class has an instance variable *pet* of type *Dog.* Note that the class diagram does not show this instance variable. You can show it; however, it is understood because of the association arrow and the role name.

1. The code implied by the diagram:

|  |  |  |
| --- | --- | --- |
| **public** **class** Person {  **private** Dog pet;  } |  | **public** **class** Dog {  } |

1. **Object Diagram** – An object diagram shows instances of classes at run-time. It shows *objects* not classes. Objects are underlined to differentiate them from classes. It is useful sometimes to visualize what could/might exist at some point in time as a program runs. An object diagram does not show abstract classes nor interfaces.



1. Example:

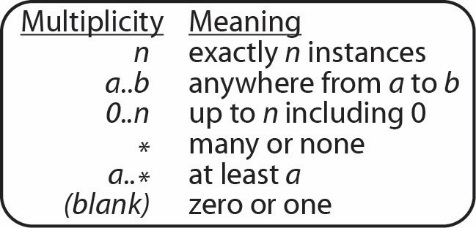
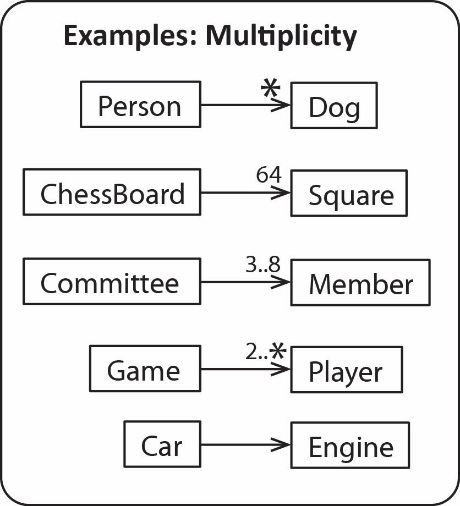
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Class Diagram |  | Code |  | Object Diagram |
|  |  |  |  |  |
| G:\eDataClasses\CS 4321\CS 4321 - Fall 18\topics\02_UML\notes\a2.jpg |  | **public** **class** Robot {  **private** Arm leftArm;  **private** Arm rightArm;  } |  | G:\eDataClasses\CS 4321\CS 4321 - Fall 18\topics\02_UML\notes\a3.jpg |

1. Example:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Class Diagram |  | Code |  | Object Diagram |
|  |  |  |  |  |
| G:\eDataClasses\CS 4321\CS 4321 - Fall 18\topics\02_UML\notes\a4.jpg |  | **public** **class** Course {  **private** Course prereq;  } |  | G:\eDataClasses\CS 4321\CS 4321 - Fall 18\topics\02_UML\notes\a5.jpg |

1. Example – A linked list is modelled with a reflexive association.

|  |  |  |
| --- | --- | --- |
| Class Diagram |  | Object Diagram (LinkedList<Dog>) |
|  |  |  |
| E:\Data-Classes\CS 4321 - Fall 2016\UML\cd8.jpg |  | G:\eDataClasses\CS 4321\CS 4321 - Fall 18\topics\02_UML\notes\a12.jpg |

1. **Multiplicity**
2. The multiplicity of an association can take on any of the values shown on the right.
3. Examples are shown on the right are called 1-to-many relationships. These are read:

* Each *Person* has many *Dogs*; each *Dog* has one *Person*. (Technically, it would be: each *Person* has zero-to-many *Dogs* and each *Dog* has zero or one *Person*; however, frequently we will use the later statement)
* Each *ChessBoard* has 64 *Squares*; each *Square* has one *ChessBoard.*
* Each *Committee* has 3-8 *Members*; each *Member* has one *Committee.*
* Each *Games* has at least 2 *Players*; each *Player* has one *Game.*
* A *Car* has an *Engine,* an *Engine* has a *Car.*

1. In other words, when reading left to right:

Singular for the class on the left, multiplicity on the right class.

And when reading right to left:

Singular for the class on the right, multiplicity on the left class.

1. **Implementing 1-to-many – Example:**

|  |  |  |  |
| --- | --- | --- | --- |
| Class Diagram |  | Code | |
|  |  |  | |
| G:\eDataClasses\CS 4321\CS 4321 - Fall 18\topics\02_UML\notes\a7.jpg |  | // Can use Collection  **public** **class** Person {  **private** List<Dog> dogs;  }  // or Map where the key is *name* (assuming unique)  **public** **class** Person {  **private** Map<String,Dog> dogs2;  } | |
| Object Diagram | | |
|  | | |
| G:\eDataClasses\CS 4321\CS 4321 - Summer 19\topics\02_UML\b2.jpg | | |

1. Example:

|  |  |  |  |
| --- | --- | --- | --- |
| Class Diagram |  | *ReportManager* instance variables | |
|  |  |  | |
| E:\Data-Classes\CS 4321 - Fall 2015\Notes\Lesson05-UML\a3.jpg |  | ArrayList<QuarterlyReport> qReports;  QuarterlyReport totals;  QuarterlyReport averages; | |
| Object Diagram | | |
|  | | |
| E:\Data-Classes\CS 4321 - Software Engineering 1\Topics\Ch05-UML\Fall10\pics\e1b.jpg | | |

1. **Two-way Navigability** –
2. Technically, if no navigability is shown (diagram on the left) then there is two-way navigability: a *Person* knows his *Dog* and a *Dog* knows his owner (*Person*). I prefer to use the explicit specification of two-way navigability as shown on the right.

**G:\eDataClasses\CS 4321\CS 4321 - Fall 18\topics\02_UML\notes\a10.jpg G:\eDataClasses\CS 4321\CS 4321 - Fall 18\topics\02_UML\notes\a9.jpg**

1. Implementing two-way navigability –The preferred approach:

|  |  |  |
| --- | --- | --- |
| **public** **class** Person {  **private** Dog pet;    **public** Person(Dog pet) {  **this**.pet = pet;  }  } |  | **public** **static** **void** main(String[] args) {  Dog d = **new** Dog();  Person p = **new** Person(d);  d.setOwner(p);  } |

Thus, the assignment of navigability is explicit:

Dog d = **new** Dog();

Person p = **new** Person(d);

d.setOwner(p);

1. Implementing two-way navigability –Inferior approach:

|  |  |  |
| --- | --- | --- |
| **public** **class** Person {  **private** Dog pet;    **public** Person(Dog pet) {  **this**.pet = pet;  **this**.pet.setOwner(**this**);  }  } |  | **public** **class** Dog {  **private** Person owner;    **public** **void** setOwner(Person owner) {  **this**.owner = owner;  }  } |

Notice the *Person* constructor calls the *Dog’s setOwner* method. However, this is not the preferred solution. In general, it is an expectation that a constructor will not have side-effects; it’s job is to initialize the instance variables. To someone using the person class, it is not obvious that the *Dog’s* owner is being set. For example:

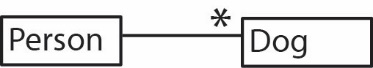
Dog d = **new** Dog();

Person p = **new** Person(d);

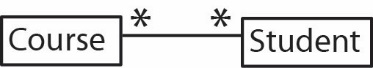
1. **Types of Associations** – In general, there are three types of associations:
2. *One-to-one* relationship: each *Person* has one *Dog,* each *Dog* has one *Person.*

E:\Data-Classes\CS 4321 - Summer 18\topics\05_UML\b1.jpg

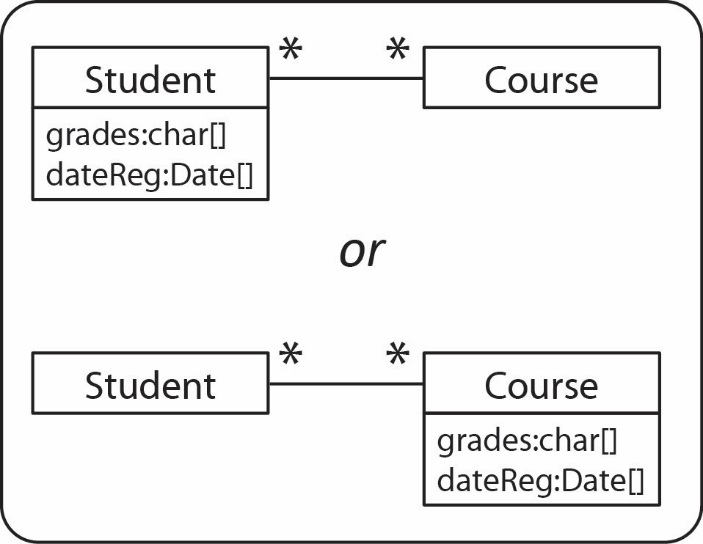
1. *One-to-many* relationship: each *Person* has many *Dogs,* each *Dog* has one *Person.*



1. *Many-to-many* relationship: each *Course* has many *Students*; each *Student* has many *Courses.*

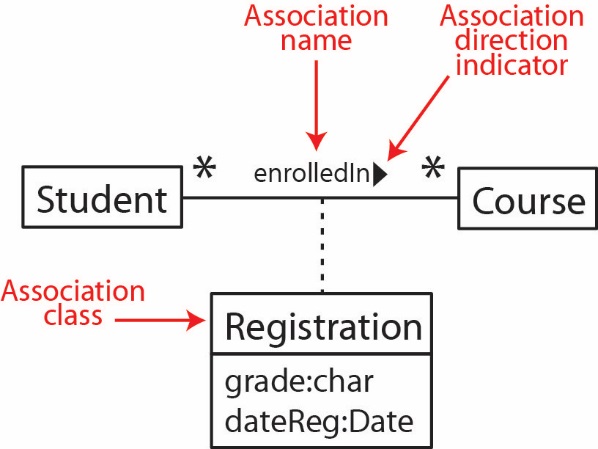
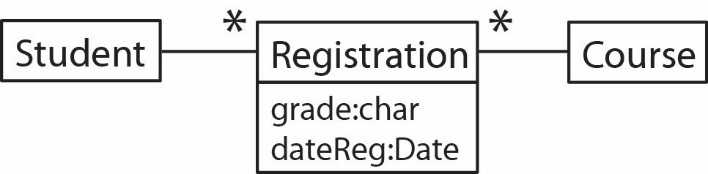


1. E:\Data-Classes\CS 4321 - Software Engineering 1\Topics\Ch05-UML\Fall12\pics\aa6.jpg**Implementing many-to-many** – The example on the right is a *many-to-many* relationship. In the early stages of modelling we might model this situation like this. However, we rarely implement *many-to-many* relationships as we did with one-to-many associations, as they are complex to manage. For example, which class do grades belong in? Which class does the date registered go in?

Notice in the top figure on the right that *grades* is a *char* array representing the student’s grade for each course she is enrolled in. To illustrate the complexity, suppose the *Course* class needs a method to calculate the gpa for the entire course. The *Course* knows (has a reference to) all the *Students,* but which grade from the *grades* array would it pull? Of course we could use *Maps* to hold the grades and dates registered. However, these parallel structures are less than ideal

In the bottom figure we have placed the *grades* array in the *Course* class. Suppose the *Student* class needs a method to calculate her gpa. She can access each course she is enrolled in, but which grade would she choose from the Course’s grades array?

The solution is almost always to introduce another class to make the implementation simpler.

1. **Association Class**
2. In the situation above we should realize that a student’s grade for a course is not exclusively a property of the *Student* nor the *Course*; it is a property of the relationship between a *Student* and a *Course*. Thus, there is an implicit class, called an *association class* that is a product of this relationship which is shown in the figure on the right. There is an instance of the *Registration* class for each instance of a *Student* being associated with a *Course.*
3. What this means in terms of implementation is shown in the class diagram on the right. We read this diagram:

* Each *Student* has many *Registrations* and each *Registration* is associated with exactly one *Course.*
* Each *Course* has many *Registrations* and each *Registration* is for exactly one *Student.*

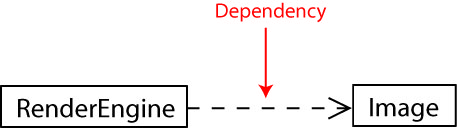
1. Either of the two representations above is acceptable. I prefer the second one.
2. An object diagram of this situation is:



Actually writing the code to implement this takes a bit of thought. This is left as an exercise for the reader (read: your project!).

1. **Association Class** – Another example. A *Person* has many *Flights*, each *Flight* has many *Persons*.

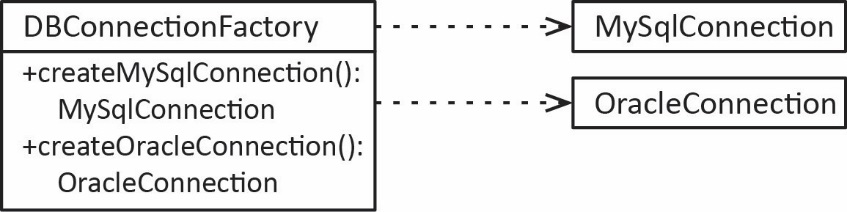
|  |  |  |
| --- | --- | --- |
| **E:\Data-Classes\CS 4321 - Summer 18\topics\05_UML\c1.jpg** |  | **E:\Data-Classes\CS 4321 - Summer 18\topics\05_UML\c2.jpg** |

1. **Dependency –**
2. A dependency exists between two classes when one class uses another class and is depicted as a dashed arrow with an angle bracket on the end pointing to the dependency as shown in the diagram on the right.
3. A dependency exists when:

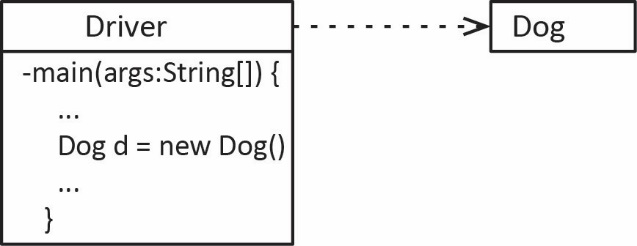
* An instance of one class is passed as an argument to a method in another class, but that class doesn’t own the instance (as an instance variable), it simply uses it.



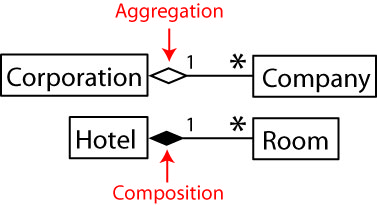
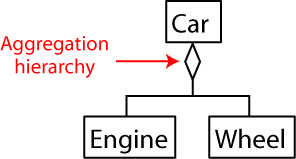
* A method in one class creates an instance of another class and returns it, but doesn’t posses one as an instance variable.



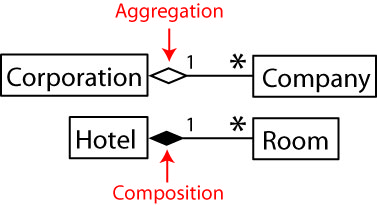
* A method in one class creates an instance of another class and simply uses it. For example, frequently, a driver class (or test class) will have a *main* that instantiates classes and uses them, but doesn’t posses them as instance variables.

****

1. An association is a strong form of dependency where one class holds a reference to an instance of another class and is depicted as a solid line, possibly with an arrow on an end(s), as discussed earlier.
2. There are usually many dependencies in a system. I suggest using them only when you need to illustrate a particularly important dependency and many times not at all.
3. **Aggregation** – This is a stronger form of association which indicates a *part-whole* relationship. In the example below, a *Corporation* is an aggregate of a number of *Companies.* This has no implication for implementation; in other words, it is implemented the same as an association. This should be used sparingly, only when it is important to illustrate a stronger association. Sometimes we use aggregation to illustrate an *aggregation hierarchy.*

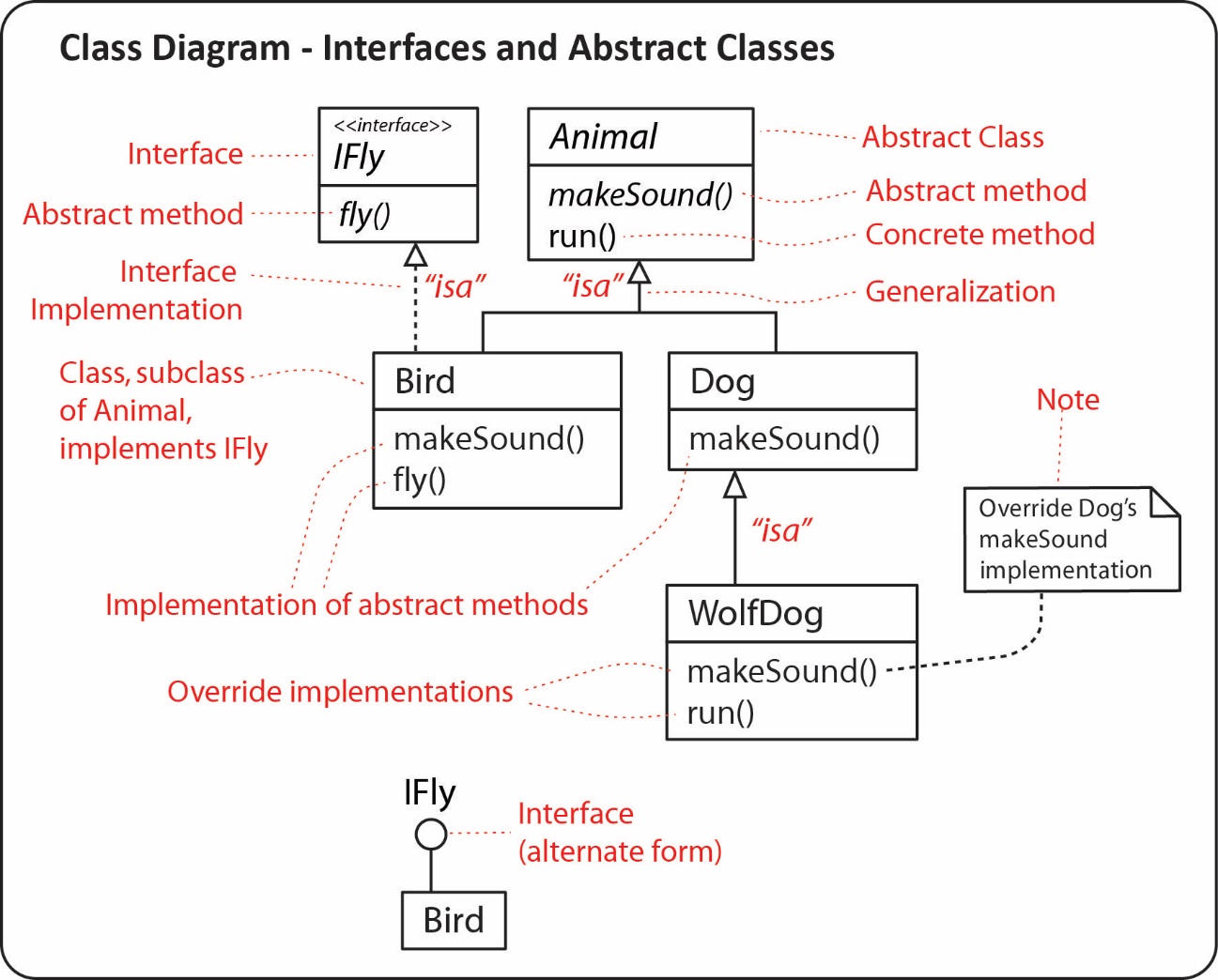
** **

1. **Composition** – This a stronger form of aggregation. It means that the *parts* cannot exist without the *whole*. For example, in the figure below, a Room object cannot exist unless it is associated with a Hotel. Composition is usually implemented so that the *whole* creates the *parts*. In the example below, the *Hotel* would create the *Room* objects.

****

1. **Abstract Classes and Interfaces.** The figure below illustrates these ideas. Note the following:

* An abstract class name and abstract methods are always italicized. On handwritten homework, or a test, surround the name with double quotes – please, don’t try to draw italics.
* A class that extends another class is depicted by showing a solid line with an open triangle on the end pointing to the superclass.
* An interface name is usually italicized, though some authors do not. Sometimes an interface name has an <<interface>> stereotype
* A class that implements an interface is depicted by showing a dashed line with an open triangle on the ending pointing to the interface.
* An alternate form for depicting an interface is with an open circle with the name above. This is used when it is not important to display the methods in the interface.
* A class that is extending another class, or implementing an interface, should show in non-italics the abstract methods it is implementing.
* A note can be added to any diagram by enclosing the text in a box and drawing a dashed line (as shown in the figure below).
* Abstract classes and interfaces are not shown in an object diagram.

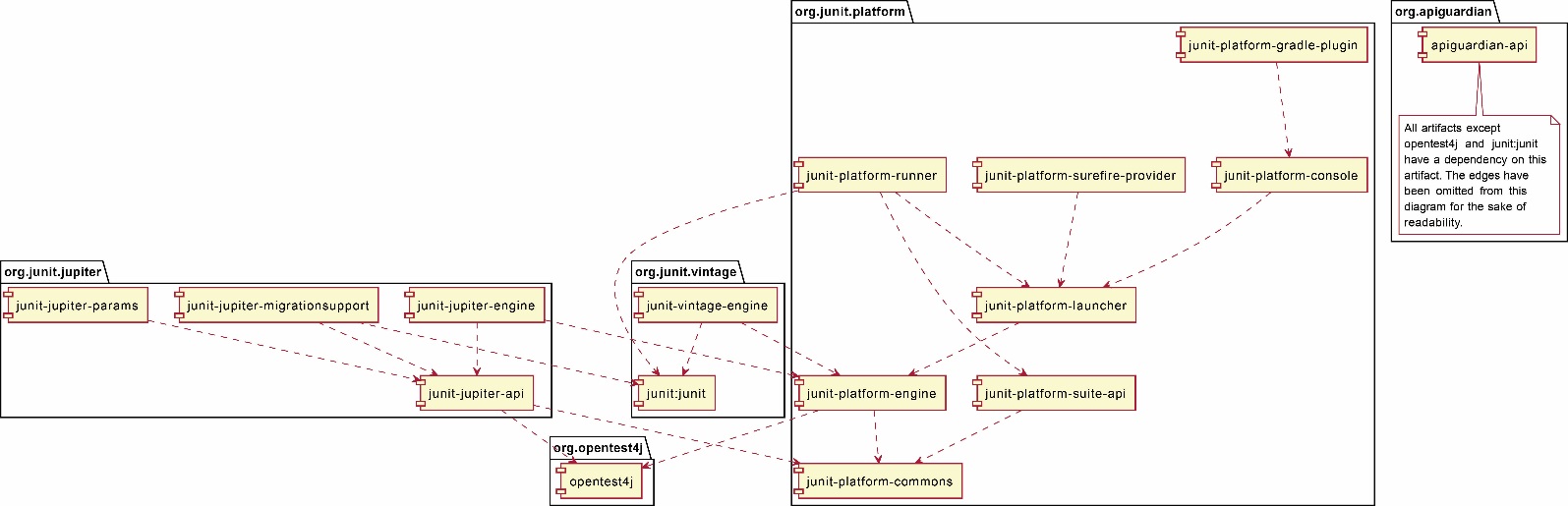
****

**Package Diagrams**

1. A package is depicted as a larger rectangle with a smaller rectangle on top which contains the name of the package. The larger rectangle may show the classes it contains, for example, the *domain, ui,* and *common* packages shown on the figure on the left. Usually, the contained classes are not shown, as in the figure on the right. In this case, there would be a separate class diagram for each package. A dashed arrow indicates that one class is importing (dependency) the classes in the package pointed to by the arrow.

|  |  |  |
| --- | --- | --- |
| Example 1 |  | Example 2 |
| E:\Data-Classes\CS 4321 - Software Engineering 1\Topics\Ch05-UML\Fall10\pics\e4.jpg |  | **E:\Data-Classes\CS 4321 - Software Engineering 1\Topics\Ch05-UML\Fall10\pics\e5.jpg** |

Example 3 – JUnit 5 Dependency diagram – The figure below technically shows *components* (rectangle with two protruding boxes). A component is a mechanism for the logical organization of code. Each component may contain multiple packages (or classes), etc. We will study these later.



Source: <https://junit.org/junit5/docs/current/user-guide/#dependency-diagram>

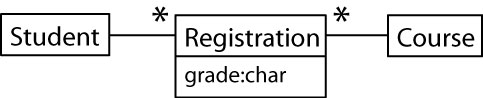
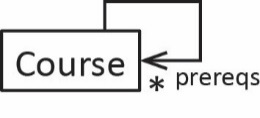
**Homework**

1. Draw the exact UML symbol, notation to represent: classes, associations, role names, association names, attributes, operations, generalizations, abstract classes, interfaces, dependency, packages, multiplicity, navigability, association class, reflexive association, aggregation, composition.
2. (a) Draw a UML Class Diagram that represents the following situation. There is an abstract class A with a public instance variable x of type integer and with a private instance variable y of type integer. This class has an abstract method m1 that takes no arguments and returns an integer. It also has a public, concrete method m2 which takes a single integer argument and doesn’t return anything. Class B is a concrete subclass of A. Class C implements the D interface which specifies a single method, display. Class C also maintains a list of three A objects and a link to an E object. Class E has a link to another object of class E. Any E object can also reference a C object. (b) Draw a UML Object Diagram for the situation described above. \*Solution at very end.
3. (a) Draw a UML Class Diagram that represents the following situation. There is a class A with an ArrayList of B objects. Class B is abstract with subclasses C and D. Class C is composed of four D object. Also, class A is dependent on class E. (b) Draw a UML Object Diagram for the situation described above.
4. (a) Draw a UML Class Diagram that represents the following situation. There are two classes, A and B. Class A maintains a list of up to 2 objects of type B on its "left" and a list of up to 3 objects of type B on its "right." Similarly, a type B object maintains "left" and "right" lists of up to 2 and 3, respectively for different objects of type A. (b) Draw a UML Object Diagram for the situation described above.
5. Consider the code below. (a) Draw the corresponding UML Static Structure Diagram (Class Diagram). (b) Draw an object diagram representing the code in *main*.

|  |  |
| --- | --- |
| **public** **class** Driver {  **public** **static** **void** main(String[] args) {  E e1 = **new** E();  E e2 = **new** E();  D d1 = **new** D();  e2.addSub(d1);  e1.addSub(e2);  D d2 = **new** D();  e1.addSub(d2);  B b = **new** B(e1);  }  }  **public** **interface** A {  **int** m1( String s );  }  **public** **class** B **implements** A {  **protected** C myC;  **protected** B( C c ) {  myC = c;  myC.m3();  }  **public** **int** m1(String s){  **return** 3;  }  } | **public** **abstract** **class** C {  **protected** **void** m2(){  System.***out***.println("hi");  }  **public** **abstract** **void** m3();  }  **public** **class** D **extends** C {  **public** **void** m3() {  System.***out***.println("yes");  }  }  **public** **class** E **extends** C {  **private** ArrayList<C> subs = **new** ArrayList<C>();  **public** **void** m3() {  **for**( C c : subs ) {  c.m2();  }  }    **public** **void** addSub(C c ) { subs.add(c); }  } |

1. Write the code for this system.



1. Draw an object diagram for the class diagram on the right.
2. An airline flight reservation system is being developed where customers can book flights. A customer needs to know his seat number for each flight. Model with a class diagram.
3. Draw an object diagram for the class diagram on the right.

**Homework Solutions**

Problem 2 Solution

|  |  |
| --- | --- |
| (a) | (b) |
| E:\Data-Classes\CS 4321 - Fall 2016\notes\UML\dd1.jpg | E:\Data-Classes\CS 4321 - Fall 2016\notes\UML\dd2.jpg |

Problem 3 solution:

|  |  |
| --- | --- |
| (a) | (b) |
| E:\Data-Classes\CS 4321 - Fall 2016\notes\UML\dd3.jpg | E:\Data-Classes\CS 4321 - Fall 2016\notes\UML\dd4.jpg |

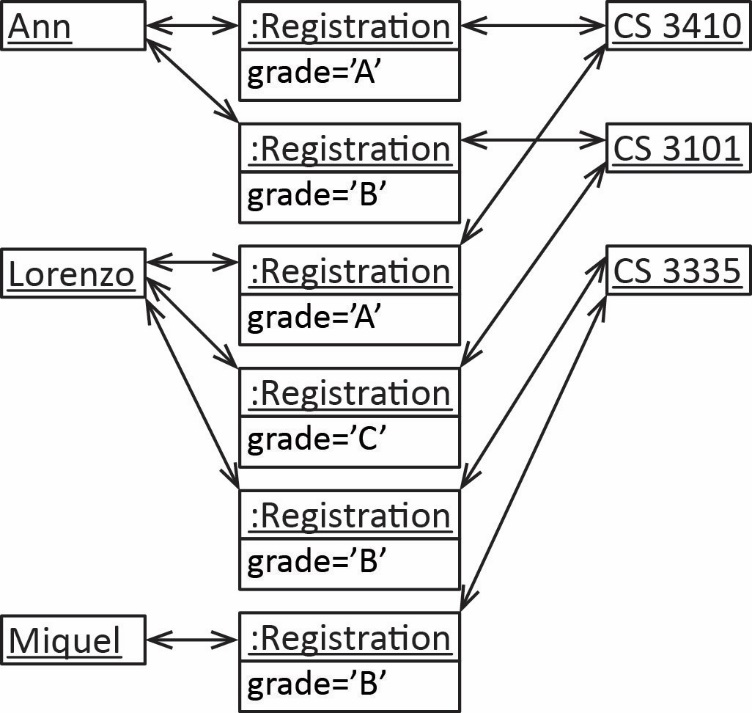
Problem 4 solution:

|  |  |
| --- | --- |
| (a) | (b) |
| E:\Data-Classes\CS 4321 - Fall 2016\notes\UML\d5.jpg | E:\Data-Classes\CS 4321 - Fall 2016\notes\UML\d6.jpg |

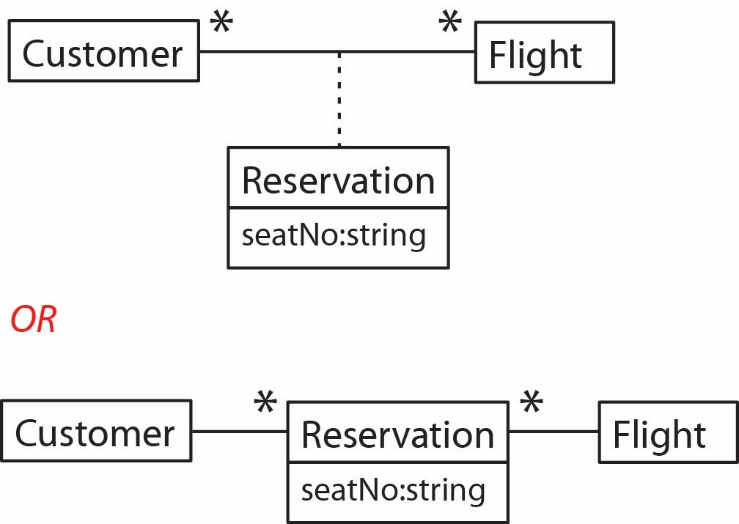
Problem 5 solution:

|  |  |
| --- | --- |
| (a)  E:\Data-Classes\CS 4321 - Fall 2016\notes\UML\dd5.jpg | (b)  E:\Data-Classes\CS 4321 - Fall 2016\notes\UML\dd6.jpg |
|  |  |

Problem 7 solution:



Problem 8 solution:



Problem 9 solution:

