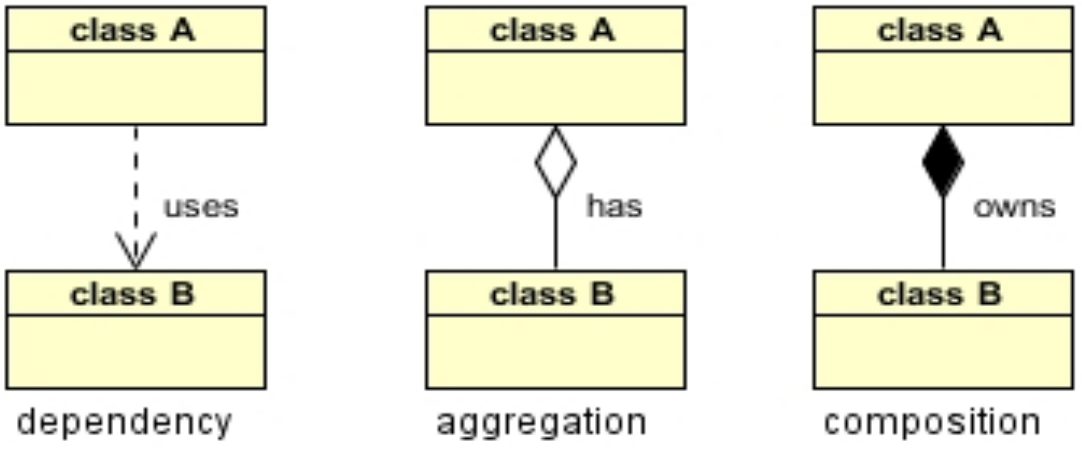
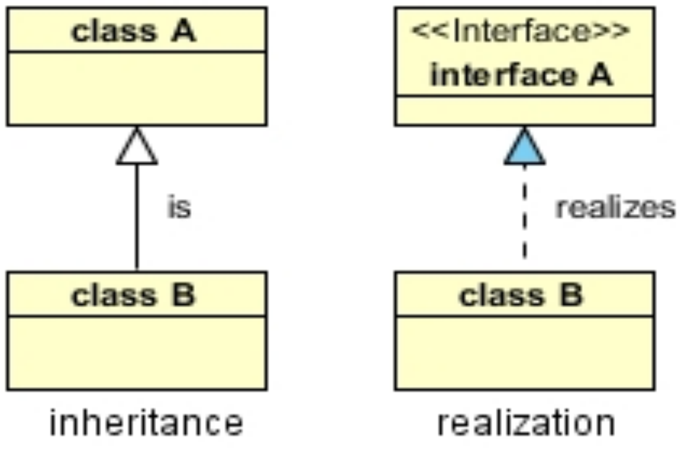
UML Class Diagram Relationships, Aggregation, Composition

There are five key relationships between classes in a UML class diagram: dependency, aggregation, composition, inheritance and realization. These five relationships are depicted in the following diagram:

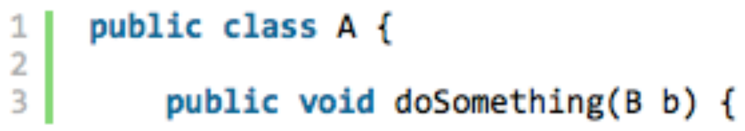




The above relationships are read as follows:

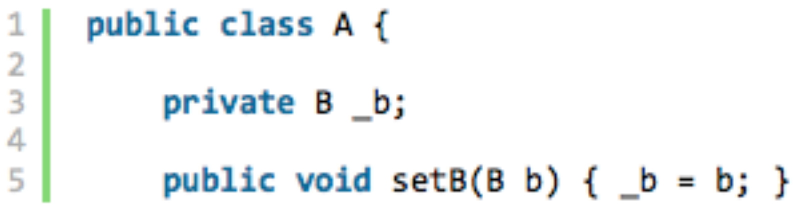
- Dependency : class A uses class B
- Aggregation : class A has a class B
- Composition : class A owns a class B
- Inheritance : class B is a Class A (or class A is extended by class B)
- Realization : class B realizes Class A (or class A is realized by class B)

Dependency is represented when a reference to one class is passed in as a method parameter to another class. For example, an instance of class B is passed in to a method of class A:



```
public class A {

    public void doSomething(B b) {
```

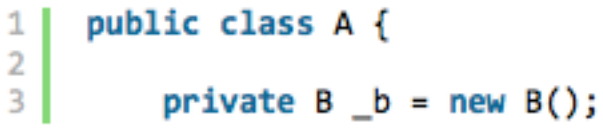
Aggregation: Now, if class A stored the reference to class B for later use we would have a different relationship called **Aggregation**. A more common and more obvious example of Aggregation would be via setter injection:



```java
public class A {

    private B _b;

    public void setB(B b) { _b = b; }
```

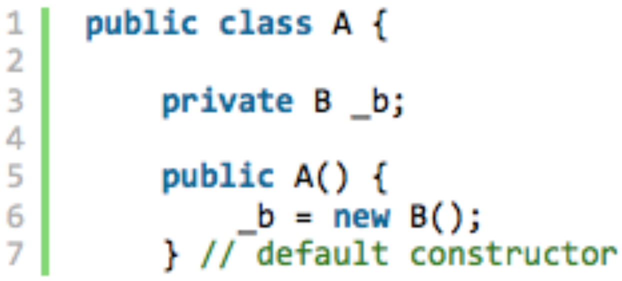
An airplane consists of wings, a fuselage, engines, flaps, landing gear and so on. A delivery shipment would contain one or more packages and a team consists of two or more employees. These are all examples of the concept of aggregation that illustrates "is part of" relationships. An engine is part of a plane, a package is part of a shipment, and an employee is part of a team. Aggregation is a specialization of association, highlighting an entire-part relationship that exists between two objects.

Aggregation is the weaker form of object containment (one object contains other objects). The stronger form is called **Composition**. The relationship modeled by aggregation/composition is often referred to as the "*has-a*" relationship where the whole and parts have coincident lifetimes, and it is very common for the whole to manage the lifecycle of its parts.. In Composition the containing object is responsible for the creation and life cycle of the contained object. Following are a few examples of Composition. First, via member initialization:



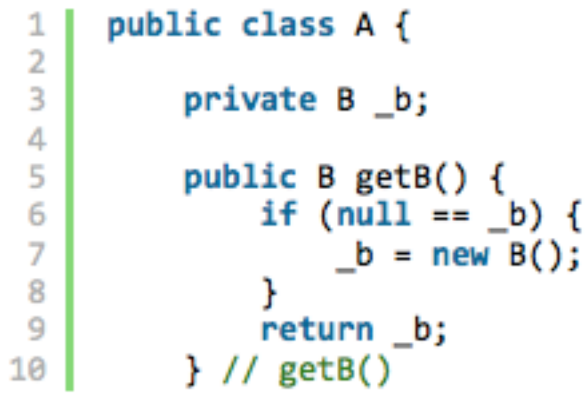
```java
public class A {

    private B _b = new B();
```

Second, via constructor initialization:



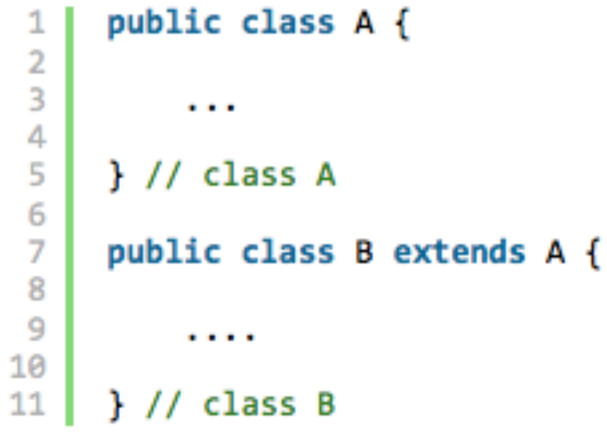
```java
public class A {

    private B _b;

    public A() {
        _b = new B();
    } // default constructor
```

Third, via lazy initialization:



```java
public class A {

    private B _b;

    public B getB() {
        if (null == _b) {
            _b = new B();
        }
        return _b;
    } // getB()
```

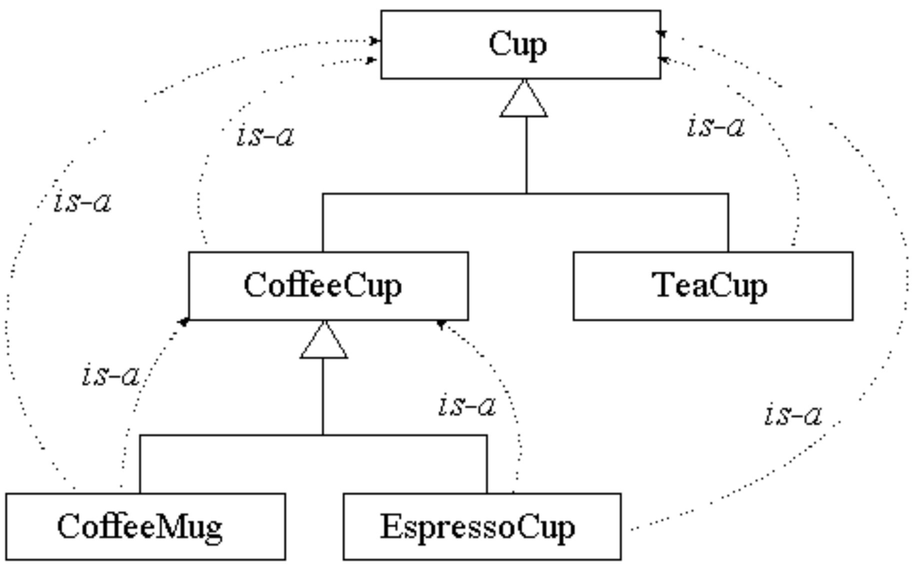
Inheritance: The relationship modeled by inheritance is often referred to as the "is-a" relationship e.g., a "`CoffeeCup` ***is-a*** `Cup`". **Inheritance** is a fairly straightforward relationship to depict in Java:



```java
public class A {

    ...

} // class A

public class B extends A {

    ....

} // class B
```

Inheritance models "is a" relationship, enabling you to rather conveniently reuse data and code that already exist. When "A" inherits from "B" we say that "A" is the subclass of "B" and that "B" is the superclass of "A." In addition to this, we have "pure inheritance" when "A" inherits all of the attributes and methods of "B". The UML modeling notation for inheritance is usually depicted as a line that has a closed arrowhead, which points from the subclass right down to the superclass.

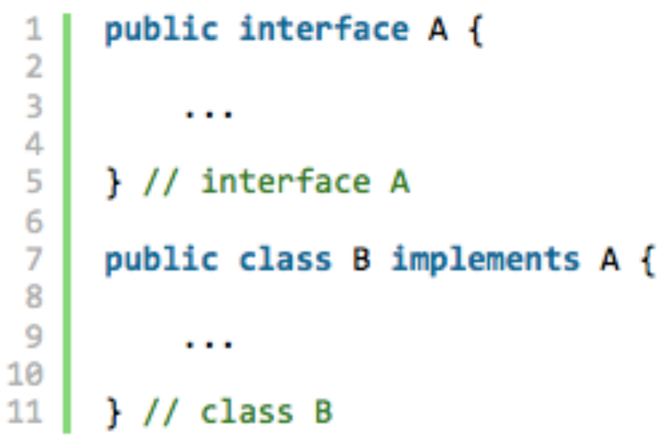
As you partition your problem domain into types you will likely want to model relationships in which one type is a more specific or specialized version of another. For example you may have identified in your problem domain two types, `Cup` and `CoffeeCup`, and you want to be able to express in your solution that a `CoffeeCup` is a more specific kind of `Cup` (or a special kind of `Cup`). In an object-oriented design, you model this kind of relationship between types with *inheritance*.



When programming in Java, we express the inheritance relationship with the *extends* keyword:

```
public class Cup {

…
} // class cup

public class CoffeeCup extends Cup {

…
} // class CoffeeCup

public class CoffeeMug extends CoffeeCup {
…

} // class CoffeeMug
```

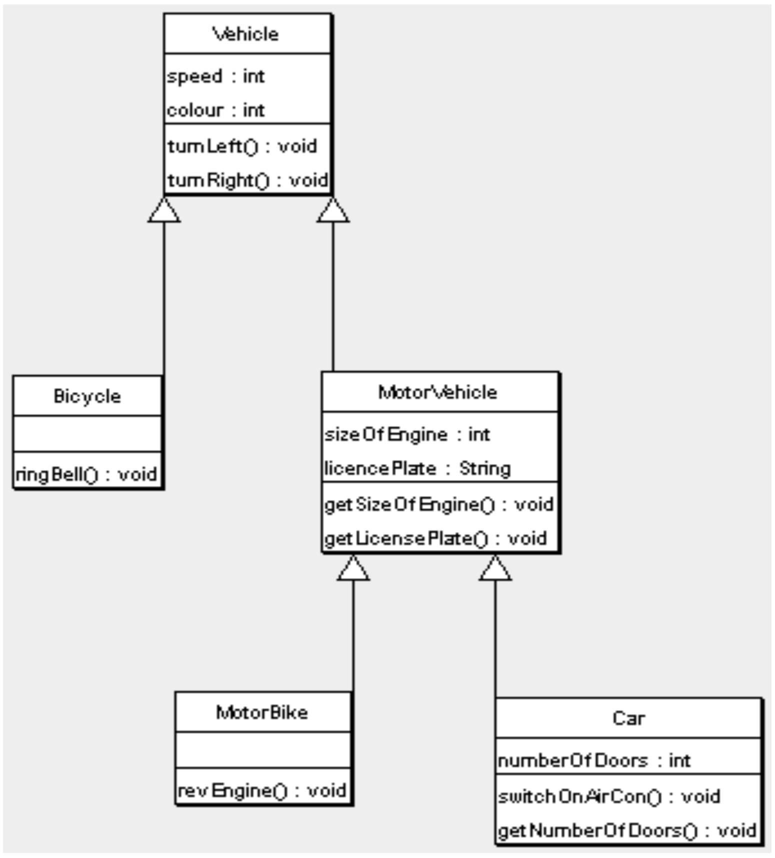
Interfaces are implemented, “**realized**” in UML parlance, by classes and components. **Realization** is also straightforward in Java and deals with implementing an interface:



```
public interface A {

    ...

} // interface A

public class B implements A {

    ...

} // class B
```

Example: Inheritance – Vehicles

This diagram shows an *inheritance hierarchy* – a series of classes and their subclasses. It's for an imaginary application that must model different kinds of vehicles such as bicycles, motorbike and cars.



Notes

- All Vehicles have some common attributes (speed and color) and common behavior (turnLeft, turnRight)
- Bicycle and MotorVehicle are both kinds of Vehicle and are therefore shown to inherit from Vehicle. To put this another way, Vehicle is the superclass of both Bicycle and MotorVehicle
- In our model MotorVehicles have engines and license plates. Attributes have been added accordingly, along with some behavior that allows us to examine those attributes
- MotorVehicles is the base class of both MotorBike and Car, therefore these classes not only inherit the speed and color properties from Vehicle, but also the additional attributes and behavior from MotorVehicle
- Both MotorBike and Car have additional attributes and behavior which are specific to those kinds of object.