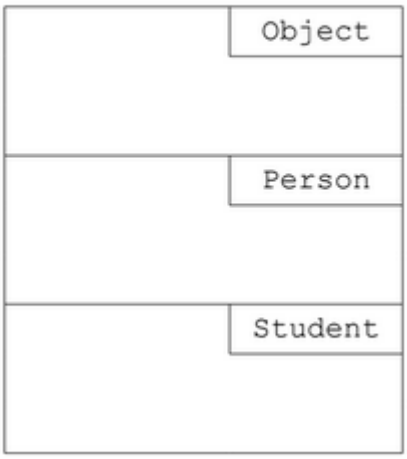
Object Interaction in Java – Inheritance & Variables

Because of the inheritance relationship in a class hierarchy, we have seen that objects can have many parts, where each part is a data type matching a different class. This forces us to reconsider our concept of a data type, and also examine how fields are inherited by objects.

Data Types and Cast

Example 1 – Consider a `Student` object. Due to inheritance, the `Student` object actually contains parts consisting of three different types (`Object`, `Person`, and `Student`).

A Student Object



As a result, the following Java statements are *all valid*.

```
Student s = new Student();
Person p = new Student(); // because a Student has a Person part
Object o = new Student(); // because a Student has an Object
part
```

This new possibility makes it critical that we think carefully about assignment statements when dealing with objects. It is possible to make assignment statements within the same hierarchy, but there are also restrictions and limitations. Consider the following statements:

Valid – the `Student` object has an `Object` part, so this assignment is legal

```
Student s = new
Student(); Object o = s;
```

Invalid (compiler error) – the `Object` object does <u>not</u> have a `Student` part, so this assignment is illegal

```
Object o = new Object();
Student s = o;
```

Things can get even more confusing when we start by mixing types in a legal manner, followed by an assignment statement.

```
Object o = new Student(); // legal because Student has an Object
part Student s = o; // error
```

In this case, an error occurs because the `Object` class does not have a `Student` part. In this case, however, because the `Object o` was created using the `Student` class, it does actually have a `Student` part. Thus we have a compiler error, even though we know the student part is

actually there.

Object Interaction in Java – Inheritance & Variables

We have used a *cast* statement in the past to force changes between data types. For example, the most typical change might be to cast a float to an int to truncate the decimal component. Another typical use might be to save a `float` as a `double` (increasing the precision of a number is never a problem, as it just adds extra zeros).

```
int a = (int)3.1415;
double x = (double)(3.2 * 5.4);
```

In terms of the compiler, the cast is an override, where the programmer is reassuring the compiler that he or she knows what they are doing, and the compiler should just ignore that particular concern.

Getting back to our objects, we can, as the programmer, recognize the logic and legality of our previous assignment statement, and override the complier's judgement with a cast.

```
Object o = new Student(); // legal because Student has an Object
part Student s = (Student)o; // ignore the error, treat o as a
Student
```

The cast is a very powerful tool in the hands of a competent programmer, but can be dangerous for a novice (as well as many experienced programmers). "With great power comes great responsibility." Consider the following modification, which also uses the cast to silence the compiler error.

```
Object o = new Object(); // o has no Student part this time Student s
= (Student)o; // ignore the error... but it's a real error!
```

Although this fragment of code will compile (since we have silenced the compiler's complaint), it will *throw an exception* at runtime, which is just another way of saying it will crash... but with a little dignity.

Inheriting Fields

Most, if not all, of the fields we have discussed thus far were declared as `private`, meaning they were only visible within their own class. This still holds true even for classes in the same hierarchy. To work with a `private` field in a different class, but in the same hierarchy, you must use *accessor* and *mutator* methods provided by the class.

For those situations where you want subclasses to have direct access to the fields of a class above them in the hierarchy, you can use the `protected` modifier in place of `private`. This makes the field visible to any class which inherits the class (i.e., subclasses), but not above it in the hierarchy, nor outside the hierarchy.

Example 2 – Consider the following class definition.

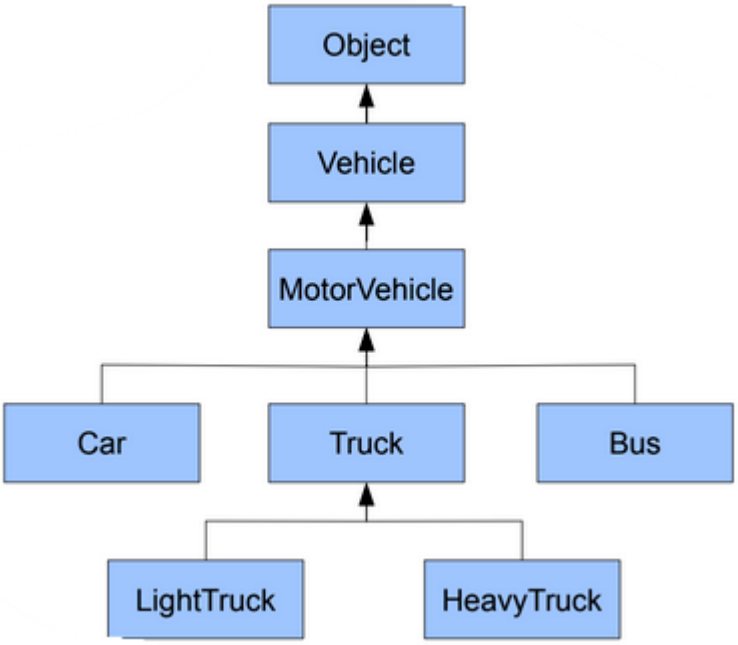
```
class Sample
{
      public int a; // visible everywhere

      float b; // visible only in the package (a collection

                // of classes) where Sample is defined

       protected char c; // visible within Sample and any subclasses
```

```
    private boolean d; // visible only within Sample
}
```

Object Interaction in Java – Inheritance & Variables

Exercises

1. Consider the class hierarchy shown here and the statements that follow it. State, with reasons, which of the statements are valid and which are not (assuming that the appropriate constructors exist for each class).



a) `Vehicle v = new Car();`

This statement is valid because the Car class contains properties that the Vehicle class contains.

b) `Truck t = new Truck();`

This statement is valid because Truck contains its own properties.

c) `Bus b = new MotorVehicle();`

This statement is invalid because the MotorVehicle class does not contain the Bus objects parts

d) `Object o = new Truck();`

This statement is valid because the Truck class contains properties that

the Object class contains.

e) `Car c = new Bus();`

This statement is incorrect because Bus is not the superclass of Car and it may contain different properties as Car class.

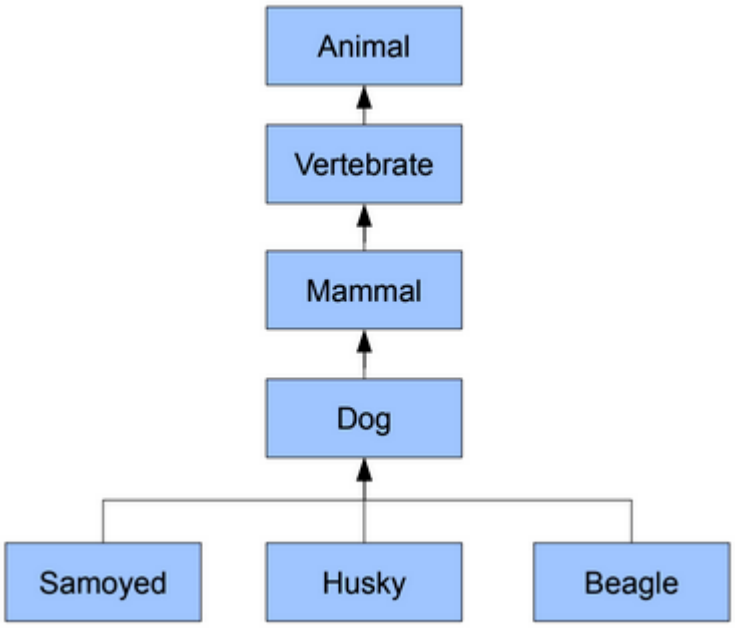
f) `Bus b = new Vehicle();`

This statement is incorrect because the Vehicle class does not contain Bus object parts.

g) `MotorVehicle m = new LightTruck();`

This statement is correct because the LightTruck class contains parts that the MotorVehicle object contains

2. Referring to the classes shown (from Example 1 of the previous lesson), which of the following conversions would require a cast?



Samoyed Husky Beagle

a) `Dog` to `Husky`

This conversion would not require a cast.

b) `Mammal` to `Animal`

This conversion will require a cast.

```
Animal a = new Mammal();
Mammal m =(Mammal) a
```

c) Dog to Animal

This conversion will require a cast.

```
Animal a = new Dog();
Dog d = (Dog) a;
```

d) Vertebrate to Beagle

This conversion will not require a cast.

```
Vertebrate v = new Beagle();
```

3. Using the classes shown in Q#2, suppose we construct the following variables.

```
Mammal pet = new Dog();
Dog toby = new Samoyed();
Dog sampson = new Dog();
```

Consider the following statements. For each one, state whether it is valid, incorrect but fixable with a cast, or not valid and not fixable with a cast. If a cast will make the statement valid, state the cast.

(a) `Samoyed s = pet;`

This statement is invalid however it is fixable with a cast of (Samoyed).

```
Samoyed s = (Samoyed) pet;
```

(b) `Object o = sampson;`

This statement is a valid statement

(c) `Samoyed s = toby;`

This statement is invalid, however is fixed with a cast of (Samoyed.

```
Samoyed s = (Samoyed) toby;
```

(d) `Dog d = pet;`

This statement is invalid, however it can be fixed with a cast of (Dog)

(e) `Mammal m = toby;`

This statement is valid.

(f) `Samoyed s = sampson;`

This statement is invalid, however it can not be fixed by simply adding a cast

```
Dog d = new Samoyed();
Samoyed s = (Samoyed) d;
```