Head First Java: Chapter 8 Notes

August 28, 2019

Polymorphism and Object References

It is important to remember that when a variable is created to refer to an object in Java, the variable itself is *not* a representation of the object itself, rather, it is a *reference* to the object. In essence, once a variable referring to an object is declared, *two* object is "created"; the object itself, and the object reference (*i.e.* the variable.) There are implications to the type of the object and the object reference:

1. The object reference variable declared and the object created is of the same type. This is the most obvious case, for example:

   ```
   Wolf aWolf = new Wolf();
   ```

2. The object reference variable declared and the object created is NOT the same type. This is possible due to polymorphism:

   ```
   Animal aWolf = new Wolf();
   ```

   In this case, the object reference variable (`aWolf`) is of type `Animal`, the superclass of class `Wolf`, while the object created that (`aWolf`) makes reference to is of type `Wolf`.

Abstract Classes and Methods

Often superclass are created as a template for subsequent subclasses, and it may not be sensible for instance of the superclass itself to be created. An example would be while `Wolf` and `Hippo` are *concrete* classes (that is, instances of that class would be sensible to create), they can be inherited from a superclass `Animal`. An object instance of `Animal` is perhaps meaningless and insensible, and can be thought of existing only to provide a template for specific animal subclasses to inherit from. In this case, `Animal` can be flagged as an `abstract` class:

```
abstract class Animal {
    private int instvar;
    public void method1() {}
    public void method2() {}
}

abstract class Canine extends Animal {
    public void method2() {}
    public void method3() {}
}
```

```
class Wolf extends Canine {
    public void method3() {}
}
```

In the above example, `Animal` is an abstract superclass with the instance variable `instvar` and method `method1()` and `method2()`. Since `Animal` was flagged as `abstract` it can never be instantiate (*i.e.* `Animal a = new Animal();` will not compile and will raise an error) and can only be inherited. The class `Canine` is a subclass inheriting from `Animal` that overrides `method2()` and adds `method3()`. However, `Canine` is *also* flagged `abstract` and therefore cannot be instantiated. Class `Wolf` is a subclass of `Canine` (and therefore also a subclass of `Animal`), and has the same instance variable and methods as `Canine` except it overrides `method3()` with its own. However, because `Wolf` is not flagged `abstract` it is a *concrete* class and can be instantiated as expected. Note that whether or not a class is flagged `abstract` or not has no bearing on it's ability to be inherited (in contrast with classes flagged as `final` or `private`, which restricts inheritance behavior.)

Like classes, individual methods can be flagged as `abstract` as well *if the class it resides in is an abstract class*. An abstract method is declared with no code body:

```
abstract class Animal {
    private int instvar;
    abstract public void method1();
    public void method2() {}
}

abstract class Canine extends Animal {
    public void method2() {}
    public void method3() {}
}

class Wolf extends Canine {
    public void method1() {}
    public void method3() {}
}
```

Abstract methods can be declared in any abstract superclasses, but down the inheritance tree they must be fully implemented at this first instance of a concrete subclass. In the above example, `method1()` in `Animal` was declared as an abstract method. It is inherited by `Canine`, which because it is itself also an abstract class, it did not have to implement the abstract method `method1()`. However, `Wolf` being the first concrete class down this inheritance tree that inherited `method1()` it must implement it by overriding the abstract method with its own code. Failing to implement an inherited abstract method in a concrete class will raise errors during compilation.

The `Object` Superclass

The `Object` class in Java is a hidden class of which all objects inherited from. This implies that any classes created by the user can utilize polymorphism with the `Object` superclass. Some methods in the `Object` superclass that are inherited by all other objects in Java include:

1. `equals(Object o)` — Evaluates if the object the method was called upon and the object argument passed (in this case the object `o`) is equal and returns a Boolean value.
2. `getClass()` — returns the class that object was instantiate from.
3. `hashCode()` — returns a unique hash code of the object.
4. `toString()` — returns a `String` that represents the object (`name@memory-location`).

There are important nuances with polymorphism that can be illustrated with polymorphic references of type `Object`. Consider the following example with `ArrayList`:

```
ArrayList<Object> myDogArrayList = new ArrayList<Object>();
Dog aDog = new Dog();
myDogArrayList.add(aDog);
// Dog d = myDogArrayList.get(0)
```

The first three lines of the above listing is valid. The variable `myDogArrayList` was declared as an `ArrayList` parameterized to only take `Object` class elements. Since all other objects (including `Dog`) inherits from `Object`, `myDogArrayList` can hold elements that is of the class `Dog` (*e.g.* `aDog`, in this case.) However, the commented out last line of code is not valid and will not compile. This is because each objects in `myDogArrayList` are actually **object references**, not the objects themselves, and because `myDogArrayList` was declared as an `ArrayList` parameterized to only hold `Object` class elements, all **object reference** elements in `myDogArrayList` are of class `Object` and not class `Dog`. In the last lined of (commented out) code, an attempt to declare a class `Dog` variable from the class `Object` **object reference** object, which will fail at compilation because there isn't a way to automatically convert `Object` to `Dog` (this would be similar to trying to declare a variable of a primitive type with a primitive of something different, *e.g.* `int x = "1";`).

This nuance between object reference type and object type in polymorphism also manifest in arguments and return types. For example:

```
public Object getObject(Object o) {
    return o;
}
```

```
public void go() {
    Dog aDog = new Dog();
    // Dog bDog = getObject(aDog);
}
```

Here a method `getObject()` was declare to take an `Object` argument and returns the same `Object`. In another method `go()`, a `Dog` class variable `aDog` was created. This variable is an object reference to the `Dog` class object, which itself is also of the class `Dog`, was passed to `getObject()`. This is valid due to polymorphism; `getObject()` takes any `Object` as an argument, but `aDog` of `Dog` class is a subclass of `Object` and therefore can be passed into the `getObject()` method (that is, the IS-A test "a Dog IS-A(N) Object" passes.) However, the last line of commented out code is invalid. This is because the return value of `getObject()` is a class `Object` **object reference** to the `Dog` class `aDog` (not the `Dog` class itself), and attempting to declare an object of `Object` class as a `Dog` class variable is invalid.

A similar implication is methods of classes is true. Consider this example:

```
class Dog {
    void bark() {}
}

public Object getObject(Object o) {
    return o;
}

public void go() {
    Dog aDog = new Dog();
    Object bDog = getObject(aDog);
    bDog.hashCode();
    // bDog.bark();
}
```

Here it is established that the `Dog` class (which like all objects are a subclass of `Object`) has a method `bark()`, and the `getObject()` method is defined as previously. In this version of `go()`, the line `Object bDog = getObject(aDog);` avoids the previous error; the `aDog` (which is of class `Dog` and thus has the `bark()` method) is passed to `getObject` which returns an `Object` class object reference, was declared as a class `Object` variable `bDog`. The next line of code is valid; `hashCode()` is a method in all objects of `Object` class, and `bDog`, while an object reference pointing to a `Dog` class object, is of class `Object` naturally possesses the `hashCode()` method. However, the last commented out line of code is not valid, because even though `bDog` points to a `Dog` class method, it is itself an object reference method of

class `Object` and do not possess the method `bark()`, and this line of code will cause compilation to fail.

A helpful way to clarify the thinking behind this nuance is to think of an object like concentric circles. A `Dog` class object can be thought of as two concentric circle, with the inner circle of the class `Object` and the outer circle of the class `Dog`. A `Dog` class object really is just one object that is both a `Dog` and an `Object` (a Dog IS-A(N) Object). However, difference classes of object references can be created for a `Dog` class object. A `Dog` class object reference (*e.g.* `Dog aDog = new Dog();`) "points" to the outer circle, and thus has access of all the methods, variables, and other members of both class `Dog` and `Object`. However, a `Object` class object reference to the `Dog` class object (*e.g.* `Object bDog = new Dog();`) points to the inner circle, and only has access to variables/methods/members of the `Object` class. Both `aDog` and `bDog` are object references that is referencing a `Dog` class object, but has different level of access.

The above behavior described is not limited to just for the class `Object`, but in fact to all levels in inheritance and must be taken into account whenever polymorphism is utilized.

Casting

Creating an object reference to an object do not modify the object it is referencing in anyway. That means that declaring an `Object` class variable to a class `Dog` object: `Object bDog = new Dog();` does not strip the `Dog` object of its "Dog-ness". In fact the `bDog` object reference can be *casted* back to the class `Dog`:

```
class Dog {
    void bark() {}
}

public Object getObject(Object o) {
    return o;
}

public void go() {
    Dog aDog = new Dog();
    Dog bDog = (Dog) getObject(aDog);
    bDog.hashCode();
    bDog.bark();
}
```

The key line of code here is `Dog bDog = (Dog) getObject(aDog);`, which tells the compiler to *cast* the returning `Object` class object reference from `getObject()` to class `Dog`. This is possible because `getObject()` really didn't modify `aDog`, but rather returned an object

reference "pointing" to the "inner (Object) circle" of the `Dog` class object. By casting the returning object reference from `getObject()` with `(Dog)`, the object reference is now "pointing" to the "outer (Dog) circle" of the `Dog` class object. The resulting `bDog` variable thus has access to both `Object` class and `Dog` class methods (and other class members.)

Interfaces

Java has limited support to multiple inheritance (that is, a class that in inheriting from multiple superclass at the same level), and it is generally desirable to avoid that. However, it might be sometimes sensible for certain subclasses within an inheritance tree to implement the same class members, while others do not. For example, using the superclass `Animal` as an illustration again, perhaps certain concrete subclasses like `Dog` and `Cat` can benefit from having class elements that enable there usage in a program for a pet store. Like other subclasses of `Animal`, `Dog` and `Cat` has methods like `roam()` and `eat()` just like `Hippo` and `Lion`. However, it is not feasible to create another level of inheritance `Pet` because `Dog` and `Cat` already inherits from different subclasses within the `Animal` inheritance tree:

```
abstract class Animal {
    private int instvar;
    abstract public void eat();
    public void sleep() {}
}

abstract class Canine extends Animal {
    public void bark() {}
}

class Wolf extends Canine {
    public void eat() {}
    public void bark() {}
}

class Dog extends Canine {
    public void eat() {}
}

abstract class Feline extends Animal {
    public void purr() {}
}

class Lion extends Feline {
    public void eat() {}
    public void purr() {}
```

```
}

class Cat extends Feline {
    public void purr() {}
}
```

Consider the inheritance relationships above. `Cat` and `Lion` inherits from `Feline`, `Wolf` and `Dog` inherits from `Canine`, and both `Feline` and `Canine` in turns inherit from `Animal`. It is not feasible to create another level in inheritance to give `Dog` and `Cat` common "Pet" methods because they are in different inheritance sub-trees, and also it is not desirable to give `Wolf` and `Lion` "Pet" methods.

The easiest and most appropriate solution in this case is to utilize **interface**. An interface is similar to a class, except that all its methods *must* be abstract. It technically exist outside of an inheritance tree, and its methods, as they are all abstract, do not contain code body, and must be implemented by any concrete class implementing them. In this way, interfaces are similar to abstract classes, offering opportunities for templates and protocols for class members to be applied to some but not all subclasses of an inheritance tree. One way to think about this is that interface give classes *roles*. For the above example, the desired `Pet` interface can be implemented this way:

```
public interface Pet {
    public abstract void beFriendly();
    void play();
}

class Dog extends Canine implements Pet {
    public void eat() {}
    public void beFriendly() {}
    public void play() {}
}

class Cat extends Feline implements Pet {
    public void purr() {}
    public void beFriendly() {}
    public void play() {}
}
```

Both class `Dog` and `Cat` both now *implement* the interface `Pet` now. Note all methods within any interface are implicitly `public` and `abstract`; restating those keywords are redundant. By implementing an interface, a class now possesses all the abstract methods in that interface, and if the class is concrete, it must implement (that is, give the method code body) those methods because they are abstract. Unlike class inheritance, any class can implement any number of interfaces. For example, `Dog` can implement more than one interfaces like `Pet`, `ServiceAnimal`, `Omnivore`, etc. While a variable or an object

cannot be instantiated out of an interface (because it behaves somewhat like an abstract class), an array or `ArrayList` can be created to be of an interface type (or in the case of ArrayList, parameterized.):

```
public interface Pet {}

class Dog implements Pet {}

class Wolf {}

Dog aDog = new Dog();
Wolf aWolf = new Wolf();

Pet[] interfaceArray = new Pet[3];
ArrayList<Pet> interfaceArrayList = new ArrayList<Pet>;

interfaceArray[0] = aDog;
interfaceArrayList.add(aDog);

// interfaceArray[1] = aWolf;
// interfaceArrayList.add(aWolf);
```

The rules for an array or an `ArrayList` declared to be of an interface type (or in the case of an `ArrayList`, parameterized to) is similar to class polymorphism discussed previously. In the above case, `interfaceArray` and `interfaceArrayList` both can only accept elements that had implemented the interface `Pet`. This is why adding a `Dog` class element (`aDog`) is valid, but the last two line of the code (commented out) are not valid because `aWolf` is of the class `Wolf` that do not implement `Pet`.