Inheritance

In our daily lives, we classify the many things around us. We know that the world has objects like “dogs” and “cars” and “food” and we are familiar with talking about these objects as classes: “Dogs are animals that have four legs and people have them as pets and they bark, etc”. We also talk about specific objects (instances): “When I was growing up I had a beagle named Buddy. Like most beagles, he loved to hunt rabbits.”

We also talk about classes of objects at different levels. For example, take animals, dogs, and beagles. Beagles are dogs, and dogs are animals, and these “is-a” relationships between classes are very important in how we talk about them. Buddy the beagle was a dog, and so he was also an animal. But certain things I might say about Buddy make more sense in thinking of him as an animal than in thinking about him as a dog or as a beagle. For example, when I say that Buddy was born in 1966, this statement is tied to him being animal rather than him being a dog or a beagle. (Being born is something animals do in general, not something specific to dogs or beagles.) So being born is something that is part of the “definition” of a being an animal. Dogs automatically “inherit” the being-born property since dogs are animals. Similarly, beagles automatically inherit it since they are dogs and dogs are animals.

A similar classification of objects is used in object oriented programming languages. In Java, for example, we can define new (sub)classes from existing classes. When we define a class in Java, we specify certain fields and methods. When we define a subclass, the subclass inherits the fields and methods from the (higher) class. We also may introduce entirely new fields and methods into the subclass. Or, some of the fields or methods of the subclass may be given the same names as those of an existing class, but maybe we change the body of a method. We will examine these choices over the next few lectures.

Terminology

If we have a class Dog and we define a new class Beagle which extends the class Dog, we would say that Dog is the base class or super class or parent class and Beagle is the subclass or derived class or extended class. We say that a subclass inherits the fields and methods of the superclass.

class Dog {
    String dogName
    String ownerName
    int serialNumber
    Date birth
    Date death
    void Dog(){ .. }
    :
    void bark(){
        System.out.println("woof");
    }
}

1
When we declare the `Beagle`, `Doberman`, `Terrier` classes, we don’t need to re-declare all the fields of the `Dog` class. These fields are automatically inherited, because of the keyword `extends`. We also don’t have to re-declare all the methods. We can redefine them though. For example, we have redefined the method `bark` for the sub-classes above. The method `bark` in the subclasses is said to **override** the method `bark` in the `Dog` superclass. More on that below.

**Overriding ≠ overloading**

Overriding a method (namely a subclass redefines a method that is defined in its superclass) is different from overloading it. When a subclass method overrides the superclass method, the signatures of the two methods are the same, by definition – namely the method name, return type, and the types of formal parameters are the same. When you **overload** a method, the method name is the same but signature changes, namely the type and/or number and order of formal parameters changes (and possibly the type of the return value changes). Overriding can only occur from a child class (subclass) to parent class (superclass), whereas overloading can occur either within classes, or between a child and parent class.

**Overloading within a class**

Let’s first consider overloading of a method **within a class**. You should have seen examples of this in COMP 202, but I will briefly review it here. We take the example of a constructor method. When a class has multiple fields, these fields are often initialized by parameters specified in the constructor. One can make different constructors by having a different subset of fields. For example, if I want to construct a new `Dog` object, I may sometimes know the dog’s name and sometimes I may know the dog’s name and the owner’s name, and sometimes neither, so I use different constructors in each case.
public Dog(String dogName, String ownerName){
    :
}
public Dog(String dogName){
    :
}
public Dog( ){
}

The last of these constructors is the default constructor which has no parameters. In this case, all numerical variables (type int, float, etc) are given the value zero, and all reference variables are initialized to null.

Finally, notice that the following constructors are actually the same and including the both will generate a compiler error. i.e. The parameter identifiers (ownerName vs. dogName) are not part of the signature. Only the parameter types matter for deciding if two methods have the same signature.

public Dog(String ownerName){
    :
}
public Dog(String dogName){
    :
}

Overloading between classes

We can use the term overloading if we have a method that is defined in a subclass and in a superclass, namely if the signatures are different. Such a situation can easily arise. The subclass will often have more fields than the superclass and so you may wish to include one or more of the these new fields as a parameter in the method’s signature in the subclass. Or one of these new fields might replace one of the fields in the signature from the superclass. We will see examples of overloading in the lectures.

Constructor chaining

When an object of a subclass is instantiated using one of the subclass’s constructor, the fields of the object are created and these fields include the fields of the superclass and the fields of the superclass’es superclass, etc. This is called constructor chaining. How is it achieved?

The first line of any constructor is

super(...);  // possibly with parameters

If you leave this line out as you have done in the past, then the Java compiler puts in the following (with no parameters):

super();
The `super()` command causes the superclass’ constructor to be executed, which typically sets
the fields of the superclass to some value and these fields and values are inherited by the subclass.
Note that the superclass has its own `super()` statement, and so on, which causes the fields of
all the ancestor classes to be initialized.

If the superclass has more than one constructor then the subclass can choose among them by
including parameters of the `super()` call to match the signature (number and types of arguments)
of the superclass constructor. In the following example, it is assumed that the class `Dog` has `String`
fields that specify the name and owner and that the class `Beagle` merely inherits these fields (but
doesn’t redefine them). So, the `Beagle` constructors might be:

```java
public Beagle(){  
    // calls Dog()
}

public Beagle(String dogName, String ownerName){
    super(dogName, ownerName);  // calls Dog(String dogName,
                                      String ownerName)
    :
}

public Beagle(String dogname){
    super(dogname);  // calls Dog(String dogname)
    :
}
```

A few notes:

- Java does not allow you to write `super.super`. So if a sub-class wants to explicitly invoke a
  method from the superclasses’ superclass, this is not possible (in Java).

- The `super` keyword is fundamentally different from the `this` keyword. `this` is a reference
  variable, namely it references the object that is invoking the method. `super` does not refer to
  an object, but rather it refers to a class, namely the superclass.

- It doesn’t make sense to talk about a subclass constructor overriding a constructor from a
  superclass, since a constructor is a method whose name is the same as the class in which
  it belongs and the name of the subclass will obviously be different from the name of the
  superclass.

This is just an introduction to how inheritance works in Java. To learn more, consult a Java
textbook such as Lewis and Loftus, or Liang. There are hundreds of these Java textbooks sitting on
the shelf in the Schulich library, crying because no one signs them out. These textbooks have lots
of good examples. See also online tutorials such as [docs.oracle.com/javase/tutorial/java/
IandI/subclasses.html](docs.oracle.com/javase/tutorial/java/IandI/subclasses.html)
The **Object class**

Java allows any class to directly extend at most one other class. The definition of a class is of one of the two forms:

```java
class MyClass

class MySubclass extends MySuperclass
```

where `extends` is a Java keyword, as mentioned above.

If you don’t use the keyword word `extends` in the class definition then Java automatically makes `MyClass` extend the `Object` class. So, the first definition above is equivalent to

```java
class MyClass extends Object
```

The `Object` class contains a set of methods that are useful no matter what class you are working with. An instantiation of any class is always some object, and so we can safely say that the object belongs to class `Object`. As stated under the `Object` entry in the Java API: the class “Object is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class.” Notice that this statement uses the word “hierarchy” and, more specifically, it could have used the term tree. Class relationships in Java define a tree. The subclass-superclass relationship is child-parent edge. When we say that “every class has `Object` as a superclass”, we mean here in that `Object` is an ancestor, namely it is the root of the class hierarchy.

The **equals( Object ) method**

In natural languages such as English, when we talk about particular instances of classes e.g. particular rooms or particular dogs, it always makes sense to ask “is this object the same as that object?” We can ask whether two rooms or dogs or hockey sticks or computers or lightbulbs are the same. Of course, the definition of “same” needs to be given. When we say that two hockey sticks are the same, do we just mean that they are the same brand and model, or do we mean that the lengths and blade curve are equal, or do mean that the instances are identical as in, “is that the same stick you were using yesterday, because I thought that one had a crack in it?”

In Java, the `Object` class has an `equals( Object )` method, which checks if one object is the same instance as the other, namely if `o1` and `o2` are declared to be of type `Object`, then `o1.equals(o2)` returns true if and only if `o1` and `o2` reference the same object. For the class `Object`, the `equals()` method does the same thing as the “==” operator: it checks if two referenced objects are in fact the same instance.

For many other classes, you will want to overload the `equals()` method, namely use a less restrictive version of the `equals` method. You may want `x.equals(y)` to return true if and only if the objects are of the same class (i.e. type) and certain but perhaps not all fields of the objects are the same. (We met this issue last lecture with `Rectangle` and we will meet it again.) Note that I say overloading rather than overriding here. In the `HockeyStick` class, we might define an `equals(HockeyStick)` method, which might specify that two hockey sticks are equal if they have the same length.

---

1 C++ allows for **multiple inheritance**, that is, a class can extend more than one superclass. This leads to complications, for example, if two superclasses have a method with a common name, which one gets inherited?
public equals(HockeyStick stick) {
    return (this.length == stick.length);
}

Another example of the subtleties with the equals() method which you are familiar with is in the String class. You were probably told in COMP 202 that, when comparing String objects, you should avoid using the == operator and instead you should use equals(). The reason is that the == operator for String objects can behave in a surprising way. Here are a few examples:

```java
String s1 = "sur";
String s2 = "surprise";
System.out.println("sur" + "prise") == "surprise"); // true
System.out.println("surprise" == new String("surprise")); // false
System.out.println("sur" == s1); // true
System.out.println((s1 + "prise") == "surprise"); // false
System.out.println((s1 + "prise").equals("surprise")); // true
System.out.println((s1 + "prise") == s2); // false
System.out.println((s1 + "prise").equals(s2)); // true
System.out.println(s2.equals(s1 + "prise")); // true
System.out.println("surprise" == "surprise"); // true
```

This behavior is a result of certain arbitrary choices made by the designers of the Java language and it is not something you need to understand or remember. As long as you use the equals(String) method of the String class instead of == to compare Strings, you will be fine.

More generally, when you define an equals( YourNewClass ) method in a class YourNewClass, you should ensure the following:

- x.equals(x) returns true
- x.equals(y) returns true if and only if y.equals(x) returns true
- if x.equals(y) and y.equals(z) both return true then x.equals(z) returns true
- if x is not null, then x.equals(null) returns false

hashCode() method

The class Object has a method called hashCode() which returns an integer. How this integer is defined is not part of the Java specification, but often it is related to the memory address of the object. For the Java Virtual Machine running on my laptop, the hashCode() method of an object returns a 24 bit number. Test the Object.hashCode() method on Eclipse (or whatever you are using). Create a System.out.print(new Object()) and you will find that it it returns a 24 bit number, written as 6 hexadecimal symbols. See the Java API for the Object class for details. We will discuss the hashCode() method later in the course when we discuss what hashing is.
clone()

Another commonly used method in class Object is clone(). This method creates a different object, which is of the same class as the invoking object and which has fields that have identical values to those of the invoking object at the time of the invocation. Cloned objects are supposed to obey the following:

The expression ( x == x.clone() ) must return false.

The expression ( x.equals( x.clone() ) should returns true (suggested, but not required).

Note that the latter doesn’t hold for Object objects. But that’s ok, as one rarely uses Object objects and even more rarely clones them.

toString()

This method is commonly used to write out a description of an object, namely the values of its fields. As an author of the class, you are free to define toString() however you wish (as long as it returns a String).

In the Object class, the toString() method prints out the class name of the object, the @ symbol, and the hashCode of the object represented in hexadecimal. Test it out. Notice that if you define your own class e.g. HockeyStick and you don’t override the toString() method from the Object class, then your class will inherit the Object class’es toString() method. So if the variable myDog references an object from the class Doberman, then myDog.toString() might return the string say “Doberman@34a212.”

Advanced topic: class descriptors and objects

At the end of the lecture, I spent a few minutes discussing the Class class and how it relates to the Object class. Here I fill in some more of the details. This material is mostly for your interest. It is not on the final exam, and it is included here only to give you a initial sense of how it works.

First, note that a class definition includes various things: the name of the class, a list of fields and types, a list of methods and their signatures, and the instructions of each method, a reference to the superclass, and other info. When a program uses a particular class, the information about that class is stored in memory in a data structure called a class descriptor. This class descriptor is different from a .class file generated by a compiler when you compile your .java file. The class descriptor is a runtime object, which is created from the .class file. The class descriptor is not a file, but rather it is a data structure inside a running Java program. The information in a class descriptor does correspond to the information in a class file though.

Specifically, class descriptors are objects of a class called Class. Also, note that the class descriptor of the Object class, and the class descriptor of the Class class are also class descriptors. They are instances of the Class class! So, we can objects that are instances of the Object class. We can have objects (class descriptors) that are instances of the Class class.

²In my opinion, it might have been better if the folks who wrote Java called this class “ClassDescriptor”, since saying “an object of class Class” is more confusing than saying “an object of class ClassDescriptor”.

How is this useful? In lecture 23, I mentioned several methods in the `Object` class. One that I did not mention is `getClass()` which returns a reference to the class descriptor of the class that this object belongs to. So, the return type of `getClass()` is `Class`, since this method returns a reference to a class descriptor. For example, `myDog.getClass()` would return a reference to the `Dog` class descriptor, which is an object of class `Class`.

The `Class` class itself has several methods, in particular, `getSuperClass()` whose return type is `Class`. This method `getSuperClass()` returns a reference to the class descriptor of the (direct) superclass. So, if `myDog` were a `Beagle`, then `myDog.getClass().getSuperClass()` would return a reference to the `Dog` class descriptor which is an object of type `Class`.

Notice that when we consider class descriptors as objects, we have to think of objects referencing each other in two different ways:

- “instance of” - an object is an instance of a class. This defines a reference from an instance object to a class descriptor object i.e. `getClass()`.
- “subclass of” - a class descriptor (a particular kind of object, namely one that is an instance of the class `Class`) will have a reference to its superclass descriptor, i.e. `getSuperclass()` (The `Object` class is the sole exception.)

One more thing: a class is not the same thing as the descriptor of that class. A class is an abstract thing, like the set of integers. It exists in our heads, not in the computer. The descriptor of a class, though, is a concrete object in a running Java program. It is an instance of class `Class`. It corresponds to particular set of bytes/code in the memory of the computer. Whenever you see a statement like

```
Dog   myDog = new Dog():
```

you should think of two objects in memory: one is the descriptor of the `Dog` class, and the second is the newly created instance of this class. The `Dog` class descriptor must be created (loaded from the `.class` file) before the class instance can be created, since the `Dog` class descriptor contains information needed to create the `Dog` object instance.