Inheritance

In our daily lives, we classify the many things around us. 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 all 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. 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 ”super” class that it ”extends”. 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.

```java
class Dog {
    String dogName
    String ownerName
    int serialNumber
    Date birthDate
    Date deathDate
    void Dog(){ .. }
    
    void bark(){
        System.out.println("woof");
    }
}
```

last updated: 22nd Nov, 2016 at 10:26
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 is different from overloading it. When a subclass method and superclass method have the same method name and the same number, types, and order of parameters, then we say that the subtype method overrides the supertype method. When the method name is the same but the type, number, or order of parameters changes, then we say the method is overloaded. 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.

Note that the return type doesn’t play a role in either case. The reason is that, whether you are overriding or overloading, you are defining a new method and so you can define whatever return type you want. This distinction is consistent with the definition of a method’s signature: the ”signature” is the method name and the types and order of the method parameters. The return type is not part of the method signature. Thus, one overrides a method when the signature is the same, and one overloads a method when the signature changes. (Of course, understanding this distinction depends on your understanding the definition of signature!)

### Overloading within a class

Let’s first consider overloading of a method within a class. We have seen examples of this already, namely the `add` and `remove` methods of the `ArrayList` and `LinkedList` classes.

Let’s 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 only know the dog’s name. Other times I may know the dog’s name and the owner’s name, and other ties neither. So I use different constructors in each case.

```java
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.

Notice that the following constructors are the same and including them both will generate a compiler error. i.e. The parameter identifiers (ownerName vs. dogName) are not part of the method signature. Only the parameter types matter.

```java
public Dog(String ownerName){
    
}
public Dog(String dogName){
    
}
```

**Overloading between classes**

As explained above, we use the term *overloading* if we have a method that is defined in a subclass and in a superclass and 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 of the these new fields as a parameter in the method’s signature in the subclass. Or, the superclass type might be a parameter type in the superclass method, and in the corresponding subclass method we might replace the type of the corresponding parameter with the subclass type. For example, `greet(Animal a)` in the Animal class might be replaced by `greet(Dog dog)` in the Dog class.

**Constructor chaining**

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

The first line of any constructor is

```java
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’es constructor to be executed, which typically sets the fields of the superclass to some value. These superclass 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 automatically to be inherited and initialized.

The following example is a minor detail, but may be useful for helping you think about constructor chaining. Take the case of a superclass that 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. The `Dog(Place, String)` constructor could in principle use either the `Animal()` or `Animal(Place)` constructor. To specify which, we use the `super` method:

```java
class Animal {
    Place home;

    Animal() { ... }
    Animal(Place home) {
        this.home = home;
    }
}

class Dog extends Animal {
    String owner;

    Dog() { ... } // This constructor automatically calls super() which creates
    // fields that are inherited from the superclass

    Dog(Place home, String owner) {
        super(home); // Here we need to explicitly write which
        // super constructor to use.
        this.owner = owner;
    }
}
```

A few more details:

- Java does not allow you to write `super.super`. There is no way for a sub-class to explicitly invoke a method from the superclass’es superclass.

- 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.

If you want to learn more, see online tutorials [docs.oracle.com/javase/tutorial/java/IandI/subclasses.html](http://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html)

**Java 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:

```
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

```
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` (or some subclass of `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 that `Object` is an ancestor. It is the root of the class tree.

**Java equals( Object ) method – see also lecture 29**

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(Object)` method does the same thing as the “==” operator, namely it checks if two referenced objects are the same.
For many other classes, you will define a different `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 some 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:

```java
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. Note in particular the ones with `false` in the right column.

```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
```

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.

### Java `hashCode()` method

The class `Object` has a method called `hashCode()` which returns an integer, that is, a 32 bit number between \(-2^{31}\) and \(2^{31} - 1\). For the Java Virtual Machine running on my laptop, the `hashCode()` method of an object returns a positive 24 bit int, that is, a number from 0 to \(2^{24} - 1\). Test the `Object.hashCode()` method on Eclipse (or whatever you are using) by running command:

```java
System.out.println(new Object());
```

You should find that it it returns a 24 bit number, written as 6 hexadecimal symbols (base 16 with digits 0-9,a,b,c,d,e). See the Java API for the `Object` class for details.

### Java `toString()` method

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 return a `String`. 

last updated: 22nd Nov, 2016 at 10:26
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 and you don’t override the `toString()` method from the `Object` class, then your class will inherit the `Object` class’s `toString()` method. So if the variable `myDog` references an object from the class `Doberman`, then `myDog.toString()` might return a string like “`Doberman@34a212`.”

Java `clone()` method

Another commonly used method in class `Object` is `clone()`. Recall this method from Assignment 1, when you cloned large `NaturalNumber` objects, for example, in the subtraction method. In general, the `clone()` 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()` should return false.
- The expression `x.equals( x.clone())` should returns true (suggested, but not required).

These two conditions make intuitive sense. The point of cloning is to create a different object instance (first condition), but the clone is supposed to be the same as the original in whatever sense we define ”same” to mean for that object’s class.

Note that the second condition doesn’t hold for `Object` objects. But that’s ok, as one rarely clones `Object` objects.

Finally, let’s consider one subtle aspect of cloning which is that objects (such as linked lists) can reference other objects. If we clone a linked list, then do we want the list and the objects in the list to be cloned? Or do we just want to the list to be cloned but the object that the list references should not be cloned? (The former is called a ”deep copy” and the latter is called a ”shallow copy”.) There is no correct answer to this question. Its an issue of design, and what is appropriate for the situation in question.