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 the subclass "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.

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

```
class Beagle extends Dog{
   void  bark(){
        System.out.println("aaaaawwwwooooooo");
   }
}
class Doberman extends Dog{
   void  bark(){
        System.out.println("GRRRR!  WO WO WO!");
   }
}
class Terrier extends Dog{
   void  bark(){
        System.out.println(" yap! yap! yap! ");
   }
}
```

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 later.

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'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'es constructor with no parameters to be executed.

Regardless of whether they have parameters or not, superclasses may have fields and the superclass constructors may set these fields to some value. These superclass fields and values are inherited by the subclass.

Note that the superclass may have its own `super(..)` statement, or it may not – and in that case it gets the default `super()` – and so on, which causes the fields of *all* the ancestor classes automatically to be inherited and initialized.

The following example illustrates some of the details of constructor chaining. The superclass `Animal` has two constructors. The subclass `Dog` constructor chooses among them by including parameters of the `Dog` constructors `super()` calls to match the signature (number and types of arguments) of the superclass constructor. (That's just how it is done for this example, but it is not required.) Specifically, the class `Dog` has a `String` field that specifies the owner. The `Dog(Place, String)` constructor could in principle have used either the `Animal()` or `Animal(Place)` constructor. It does the latter by calling `super(home)`. (It could have done the former by not having a `super(..)` call which would have defaulted to a `super()` call.

```
class Animal {
  Place  home;

  Animal() { ... }

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

class  Dog  extends  Animal {
  String owner;
  String name;

  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`

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). 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, such as the `add` and `remove` methods of the ArrayList and LinkedList classes.

Another example is 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 may have different constructors for 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`.

Notice that the following constructors are actually 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.

```
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 overloaded by having `greet(Dog dog)` in the `Dog` class.

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

In natural languages such as English, when we talk about particular instances of classes e.g. 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 `Object` class, 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, we may want to override the `equals(Object)` method, namely use a less restrictive version of the `equals` method. We may also wish to overload it, for example, by defining an `equals(Dog)` method in the `Dog` class. For today, I will not get into why we might want to override versus overload. I'll try to say something about it in a future lecture.

We have an intuitive notion of what we mean by 'equals'. But since the *equals()* method is so fundamental in Java, the designers of the Java language specified quite formally how the method should behave. It is very similar to the mathematic definition of *equivalence classes* which you learn about in MATH 240. When writing your own classes and overriding this method, you should be aware of this. See details here:
`https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#equals-java.lang.Object-`

For example,

- `x.equals(x)` should always be true
- `x.equals(y)` should have the same true or false value as `y.equals(x)`
- if `x.equals(y)` and `y.equals(z)` are both true, then `x.equals(z)` should be true.

The `String` class overrides the `equals(Object)` method. You may have been 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 where as a user you would think the result should be true in each case. Not so: note that some cases are `false`.

```
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 should be fine.

Another example mentioned in the lecture is the Java `LinkedList` class which also overrides the `equals(Object)` method. If a `LinkedList` object invokes this method on some other object, the result will be true only if that other object is also a LinkedList object and each of the elements in the two lists are “equal”, in particular, the elements of the two lists are “equal” according to the `equals` method of these elements. The Java API gives only a short description of this condition and (in my opinion) a few details are left unspecified. Buyer beware.

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 $2^{31}-1$. The `String` class overrides this method. We have discussed this in the hashing lecture. Have a look at this definition in
`https://docs.oracle.com/javase/7/docs/api/java/lang/String.html`
and you'll see it is what we discussed.

Also have a look at the `LinkedList` class, which also overrides the `hashCode()` method.
`https://docs.oracle.com/javase/7/docs/api/java/util/AbstractList.html`
In fact this `hashCode()` method is inherited from an “abstract class called an `AbstractList`. We discuss abstract classes a few lectures from now.

[BEGIN – ADDED Nov. 20, 2017]

One important property of the `hashCode()` method is that if two objects are considered equal by the `equals(Object)` method, then they should return the same `hashCode()`. I will come back to this point a few times in the coming lectures.

[END]

`Java toString() method`

The `Object.toString()` method returns the object's `hashCode()` written as a string. The hashcode is a 32 bit integer. Interestingly, the `toString()` method doesn't convert this 32 bit `int` into the base 10 string representation you might expect. Rather it converts it into a string representation of the 32 bit int, written in base 16. This representation is known as hexadecimal. (See Appendix below.)

Specifically, the `Object.toString()` method returns a `String` object that has three parts: the class name of the object, the @ symbol, and the hexadecimal representation of the `int` returned by the `Object.hashCode()` method. Check it out by simplying testing the instruction: `System.out.print( new Object() )`.

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'es `toString()` method. So if the variable `myDog` references an object from the class `Doberman`, then `myDog.toString()` might return a string like “`Doberman@2934a212`.”

More commonly, one overrides the `Object.toString()` method and prints out a description of an object, such as 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`.

Appendix: Hexadecimal representation of binary strings

When we write down binary strings with lots of bits, we can quickly get lost. No one wants to look at 16 bit strings, and certainly not at 32 bit strings. A common solution is to use *hexadecimal*, which is essentially a base 16 representation. We group bits into 4-tuples ($2^4 = 16$). Each 4-bit group can code 16 combinations and we typically write them down as: `0,1,...,9,a,b,c,d,e,f`. The symbol `a` represents 1010, `b` represents 1011, `c` represents 1100, ..., `f` represents 1111.

Binary	Decimal	Hexadecimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	a
1011	11	b
1100	12	c
1101	13	d
1110	14	e
1111	15	f

We commonly (but not always) write hexadecimal numbers as `0x`______ where the underline is filled with characters from `0,...,9,a,b,c,d,e,f`. For example,

$$\texttt{0x2fa3} \; = \; 0010\ 1111\ 1010\ 0011.$$

Sometimes hexadecimal numbers are written with capital letters. In that case, a large `X` is used as in the example `0X2FA3`.

If the number of bits is not a multiple of 4, then you group starting at the rightmost bit (the least significant bit). For example, if we have six bits string 101011 , then we represent it as `0x2b`. Note that looking at this hexadecimal representation, we don't know if it represents 6 bits or 7 or 8, that is, 101011 or 0101011 or 00101011. But usually this is clear from the context.