

We take a pause from non-linear data structures. We will return to them at the end of the course, where we will discuss graphs. For now, let's turn to a different topic: object oriented design.

## 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 class 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, we may introduce entirely new fields and methods into the new class, or some of the fields or methods of the subclass may be given the same names as those of an existing class (but we change the body a method if we wish), or we can let the new class just inherit (unchanged) the fields or methods of the existing class. 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    name
    String    owner
    int       serialNumber
    Date      birth
    Date      death
    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 below.

## The Object class

Java allows any class to directly extend at most one other class.<sup>1</sup> 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 `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. In Java, 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

---

<sup>1</sup>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?

“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

**[I modified on April 1, by clarifying that this method has a parameter of type Object. It is useful to keep this parameter type in mind when you are overriding this method in a subclass since the signatures should match (if indeed you want to override).]**

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, 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. This is easy to do. Just check if the address is the same!

For many other classes, you will want to *override*<sup>2</sup> 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.

An example you are familiar with already is 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, some of which you may find surprising.

```
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
```

<sup>2</sup>We will say more about the term “override” below.

```
System.out.println("surprise" == "surprise");           // true
```

This behavior is a result of certain choices made in designing Java and *it is not something you need to understand or remember*. As long as you use the `equals` method instead of `==` to compare strings, you will be fine.

More generally, when you define the `equals( Object )` method in a class, and hence override the `equals( Object )` method from the `Object` class, 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 `hashCode()` which returns an integer. How exactly this integer is defined is not part of the Java specification, but often one reads that this integer is the address of the object. Now recall that I told you that memory addresses are 32 bit numbers. While this is true, it is also true that Java objects cannot just be at any arbitrary location in this address space. Rather, there is a restricted region in which objects lie (and there is a restricted region in which the instructions of the program lie, and another restricted region in which the call stack lies). For this reason, the hashCodes don't need to be 32 bit numbers. For the Java Virtual Machine running on my laptop, the `hashCode()` method of an object returns a 24 bit number.

To see this, 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 returns a 24 bit number, written as 6 hexadecimal symbols. (Recall from Exercises 1 that each hex symbol stands for a number from 0 to 15 i.e. 4 bits, so 6 hex symbols implies 24 bits. See [http://java.sun.com/javase/6/docs/api/java/lang/Object.html#hashCode\(\)](http://java.sun.com/javase/6/docs/api/java/lang/Object.html#hashCode())).

In general (including for the class `Object`), the `hashCode()` and `equals()` method are supposed to be related as follows: if `x.equals(y)` returns true, then `x.hashCode() == y.hashCode()` should return true. What about the converse, namely if `x.hashCode() == y.hashCode()` returns true then does this imply `x.equals(y)` returns true? No, it doesn't. (A collision would occur in the hash table, but that's the only problem.)

Notice that it could be disastrous if we allowed `x.equals(y)` to return true but `x.hashCode() == y.hashCode()` to return false, namely we might use a key `x` to put a value into the hash table, but if we were to use key `y` to get the value, thinking that `x` and `y` were equal, we might<sup>3</sup> not find the value.

The general rule to follow is this: if you write a class which uses the `hashCode()` method and you want to override either the `hashCode()` or `equals()` method, then you should ensure that: if `x.equals(y)` returns true, then `x.hashCode() == y.hashCode()`. This may require that you override *both* of them.

---

<sup>3</sup>We might get lucky and find it, since two different hash codes can (because of compression) get mapped to the same bucket index.

## 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:

```
x == x.clone();           // returns false (required)
x.equals( x.clone() )    // returns true (suggested, but not required)
```

Note that this behavior doesn't hold for `Object` objects though, 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`). As you can verify from the Java API, the *signature* (that is, method names, return type, and parameter types) is

```
public String toString()
```

i.e. the return type is `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's `toString()` method.

## Overriding $\neq$ overloading

Overriding a method (namely a subclass redefines a method that is defined in its superclass) is different from overloading it. When you *override* a method, the signature of the two methods are the same – 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 a method that is defined more than once *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 owner and name and sometimes I might just know its owner, and sometimes neither, so I use different constructors in each case.

```
public Dog(String name, String owner){
    :
}

public Dog(String name){
    :
}

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 will generate a compiler error. The parameter identifiers (`owner` vs. `name`) are not part of the signature.

```
public Dog(String owner){
    :
}
public Dog(String name){
    :
}
```

### Overloading between classes

We also use the term *overloading* if we have a method that is defined in a subclass and in a superclass, if the signature differs between classes. It is easy to see how such a situation might arise. The subclass will often have more fields than the superclass and so you may wish to include one or more of 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. It can also happen that you want to change what the method does for the subclass because the subclass is special. We saw examples earlier with `hashCode()`, `toString()`, `equals()`.

[**ADDED April 1**] One subtle example of overloading can occur if you accidentally define an `equals()` method in your own class, say `HockeyStick`, such that the parameter type of the `equals()` method is the class type, i.e. `HockeyStick`, rather than `Object`. For example, say two hockey sticks are equal if they have the same length:

```
public equals(HockeyStick stick){
    return (this.length == stick.length)
}
```

Note that this would overload the `equals()` method, not override it. While in most cases the distinction doesn't matter, some strange behavior can result if you think you are overriding when in fact you are overloading. This comes up in polymorphism, which we discuss next lecture.