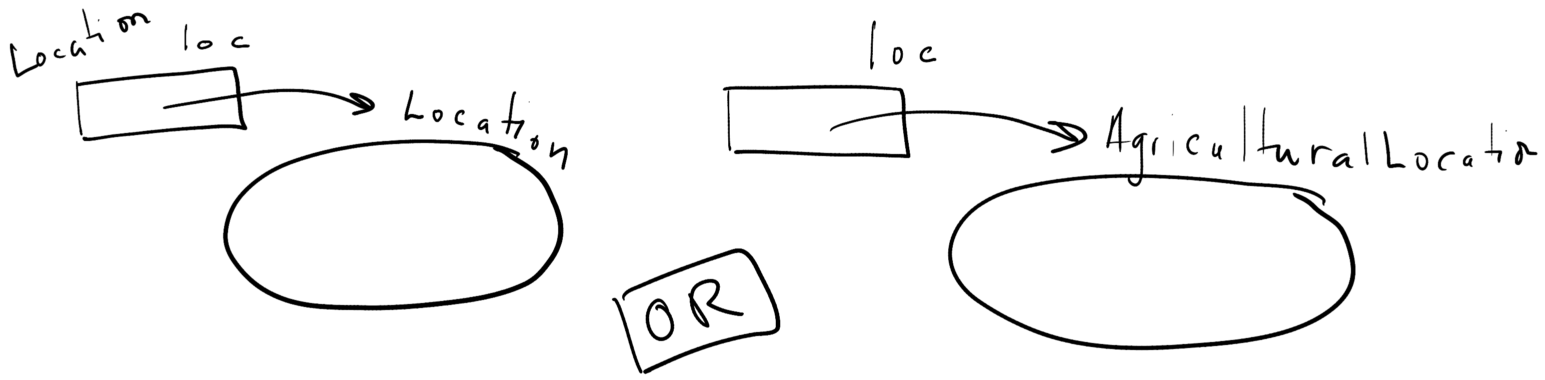
The handout will review some of the concepts we have covered related to inheritance and dynamic binding. We will use as an example two classes: Location and AgriculturalLocation (which extends Location). For brevity, the member data and signatures of the methods but not the code.

```
public class Location
{
        double longitude;
        double latitiude;
        String name;

        public Location()
        public Location( String name, double lat, double longi );
        public String toString();
        public double distanceInMeters( Location loc )
}

public class AgriculturalLocation
{
        private double avgAnnualRainfall;

        public AgrictulturalLocation( String name, double lat, double longi, double rain );
        public String toString();
        public double getAvgRainfall();
}
```

Now suppose we have declared a variable which is a reference to an instance of class location:

```
public Location loc;
```

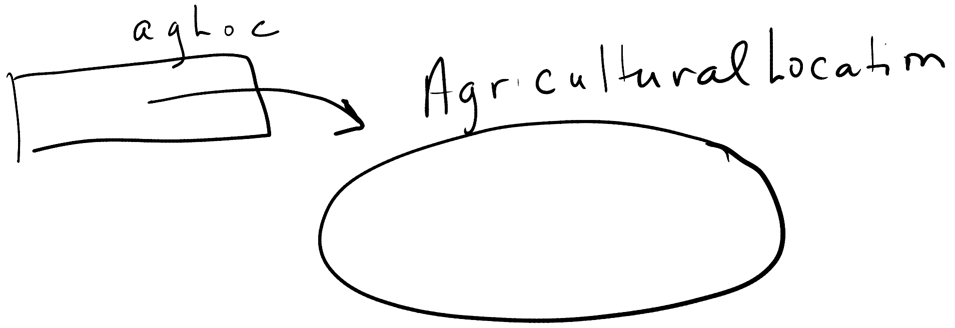
loc can refer to an instance of Location or any subclass of Location:

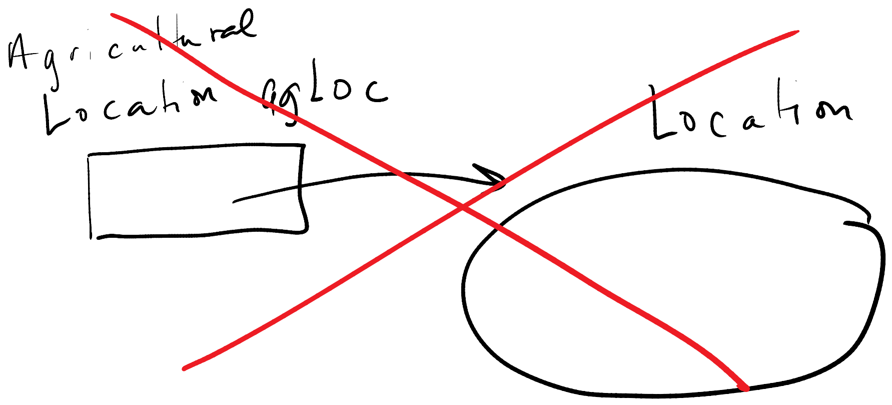


If we have a reference to instance of AgriculturalLocation:

```
public AgriculturalLocation agLoc;
```

It can point to any instance of Agriculatural Location or a subclass of Agricultural Location, but it can not point to an instance of Location which is a superclass of AgriculturalLocation:





Any assignment of something more specific to something more general is allowed:

```
agLoc = new AgriculturalLocation( “Irvine, 33, -117, 13.8);
loc = agLoc;
```

However, if we want to do perform an assignment of something more general to something more specific, then it requires a cast:

```
agLoc = (AgriculturalLocation) loc;
```

This is, in general, a dangerous thing to do. If loc happens to be pointing at an instance of AgriculturalLocation (or a subclass) then the assignment will execute without a problem. However, if it happens that loc is referring to an instance of Location, the above line will cause the program to crash at runtime. If we want to be careful and check this condition before the assignment, we can do so:

```
If ( loc instanceof AgriculturalLocation )
{
        agLoc = (AgriculturalLocation) loc;
}
```

Now, if we have a reference of type Location, then it can be used to access public member data or methods defined in Location or any of its superclasses. But the statement

```
loc.getAverageRainfall();
```

would result in a compilation error because Location does not have the method getAverageRainfall defined.

If we are sure that ***loc*** is pointing at an instance of AgriculturalLocation, we could do an explicit cast and then the method call:

```
agLoc = (AgriculturalLocation) loc;
agLoc.getAverageRainfall;
```

Now notice that we have two versions to toString() defined – one in Location and one in AgriculturalLocation. Which one gets called? This will be determined at runtime and will depend on the type of the object. This is called **dynamic binding**.

```
Location loc = new Location("BrenHall", 33, -117);
System.out.println( loc.toString() );
```

This will statement above will use the toString() defined in Location.

```
Location agLoc = new Location("Irvine", 33, -117, 13.8);
System.out.println( agLoc.toString() );
```

This will use the toString() defined in AgriculturalLocation. But what about:

```
Location loc = new AgriculturalLocation("Irvine", 33, -117, 13.8);
System.out.println( loc.toString() );
```

We have a reference of type Location, pointing to an instance of type AgriculturalLocation. When the print statement is reached, the interpreter knows the type of the object that loc is pointing to and at that moment (in this case it is AgriculturalLocation)and it finds the toString() method which is defined for instances of AgriculturalLocation. Since the class AgirculturalLocation has a toString() method defined which overrides the toString() method in Location that is the one that will be called.