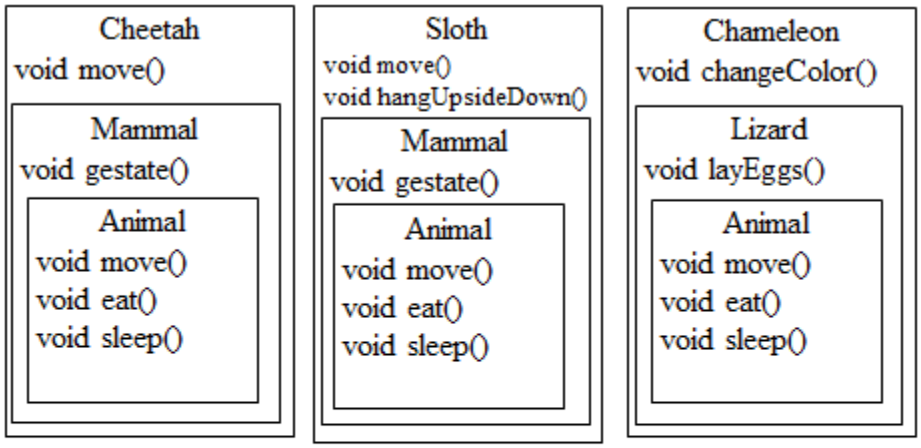
Polymorphism

If we have an inheritance hierarchy with several subclasses inheriting from a superclass, then we are guaranteed that anything public from the superclass will be available for all the subclasses. So, if we write code that uses only the public interface of the superclass, that code ought to work for the subclasses too.

For any methods that are overridden, the appropriate version of the method will be called. So, we can write code once that potentially could result in many different behaviors depending on the actual types of objects involved at runtime. Also, we can write this code purely based on the superclass, and know that any subclass -- even those only thought of and created much later – will work in this code.




```
public static void daily(Animal a) {
      a.move();
      a.eat();
      a.move();
      a.sleep();
}
```

A reference variable in Java can point to any object of its own type or of any subclass type. We can have an array of the superclass type and fill it with various instances of different subclasses. We can also have a variable of superclass type and not know which type of object it will be pointing at until runtime.

Note that since every class inherits from Object, a variable of type Object can point at any type of object. However, we will only be able to call a limited number of methods, e.g. toString, that are defined in the Object class

A reference variable of subclass type *cannot* point at an object of superclass type. Java checks what methods are legal to call at compile time, based on the type of the variable, so if we could point a subclass variable at a superclass object, it could potentially call a method that the object does not have.

.

```
Animal a = new Sloth();
a.move()
  // anything legal for Animal is
  // legal for Cheetah, Sloth, etc

Sloth s = new Animal(); // not legal
s.hangUpsideDown();     // to stop this
  // Sloth has methods Animal doesn't
```

Similarly, if a variable of superclass type is pointing at an object of subclass type, it is still only legal to call methods defined for the superclass.

```
Animal a = new Sloth();

// a points to Animal or subclass
// instance, (now: Sloth)
// and all animals have move
a.move();

// a happens to point to a Sloth now
// but not everything a can point to
// has hangUpsideDown
a.hangUpsideDown();
```

Compile Time vs Runtime

Polymorphism based on inheritance in this way is called subtyping. In Java, this kind of polymorphism is based on instance methods. For an overridden method, the version called depends on the type of an object *at runtime*.

Method overloading is also a form of polymorphism. Different versions of a method are called depending on the type(s) passed to the method. But in this case, which version of the method is called is determined at *compile time*, based on the type of the variable (or literal) passed to it. So, if there are two overloaded versions of a method, one that takes a superclass type and one that takes a subclass type, and a superclass variable is passed in, then the superclass version of the method will be called, even if the variable is actually pointing at a subclass object.

```
public void aniStuff(Animal an) {
  an.move();
}
public void aniStuff(Sloth an) {
  an.move();
  an.hangUpsideDown();
}

//...
     Animal a1 = new Sloth();
     aniStuff(a1); // calls Animal version
     Sloth a2 = new Sloth();
     aniStuff(a2); // calls Sloth version
```

When we say that variables and static members of a class cannot be overridden, only hidden, we mean that they similarly do not behave according to inheritance-based polymorphism. Which variable or static method is used is determined at compile time based on the type of the reference variable we are using to access it.

So if Animal had an instance variable x and Sloth also added an instance variable x, every Sloth object has two variables named x in it. If s is a Sloth variable pointing at a Sloth object, and we access s.x, we get the x from Sloth. If a is an Animal variable pointing at the same Sloth object and we access a.x, we get the x from Animal.

instanceof

If we have designed our classes well, we almost never need to know the actual type of the objects we are dealing with. We just call the overloaded method and get the right behavior automatically. However, it is sometimes useful to be able to check what type we are dealing with, to determine if this is a special case. The instanceof operator returns true if an object is an instance of the given type. (Vocabulary note: all subclass instances are considered to also be instances of the superclass, but not the reverse.)

Once we have used instanceof to check whether a variable is actually pointing to an object of the right type, we can use casting to point a variable of that type at the same object so that we can call methods specific to that type.

```
public void aniStuff(Animal ani) {
   // safe for all
   ani.move();
   // if actually pointing to a Sloth
   if (ani instanceof Sloth) {
      // cast and assign to Sloth var
      Sloth s = (Sloth)ani;
      // sloth variables can do this
      s.hangUpsideDown();
      // or
      ((Sloth)ani).hangUpsideDown();
   }
}
```

equals()

The Object class contains a few methods inherited by all other types. One of these is toString(). Another is the equals() method. The version of equals() in Object does exactly the same thing as ==. If we want to define a more interesting type of equality for our classes, i.e. based on the value of instance variables, then we should override equals().

```
public class Cat {
   private String name;
   private int age;
   //...

   public boolean equals (Object obj) {
      // only non-null Cats could be equal
      if (obj != null && obj instanceof Cat) {
            // cast so we can use accessors
            Cat c = (Cat)obj;
            // equal if name and age are same
            return c.getName().equals(getName())
                  && c.getAge() == age;
      }
      return false;
   }
}
```

The equals method takes type Object, so we often use instanceof inside it to check whether what we have been passed is a type for which equality makes sense. Depending on the situation, instances of one class might be considered potentially equal to instances of another class.

Note that while we can overload the equals() method so there is a version that takes the class type, this does not eliminate the need for instanceof, since which version is called will depend on variable type, not object type.

```
public class Cat {
  private String name;
  private int age;
  //…

  public boolean equals (Cat obj) {
     return c.getName().equals(getName())
            && c.getAge() == age;
  }

   public boolean equals (Object obj) {
     return false;
   }
}
```

```
Animal c = new Cat("Fluffy", 3);
Animal d = new Cat("Fluffy", 3);
if (c.equals(d)) {
//…
```

calls Object version, since variable d is not of type Cat

Splitting the logic up into different methods for the different types still makes sense, we just need to ensure that the Object version can handle any type of object it gets.

```
public class Cat {
  private String name;
  private int age;
  //…

  public boolean equals (Cat obj) {
       return c.getName().equals(getName())
              && c.getAge() == age;
  }

   public boolean equals (Object obj) {
       if (obj instanceof Cat) {
             return equals((Cat)obj);
       }
       return false;
   }
}
```

```
Animal c = new Cat("Fluffy", 3);
Animal d = new Cat("Fluffy", 3);
if (c.equals(d)) {
//…
```