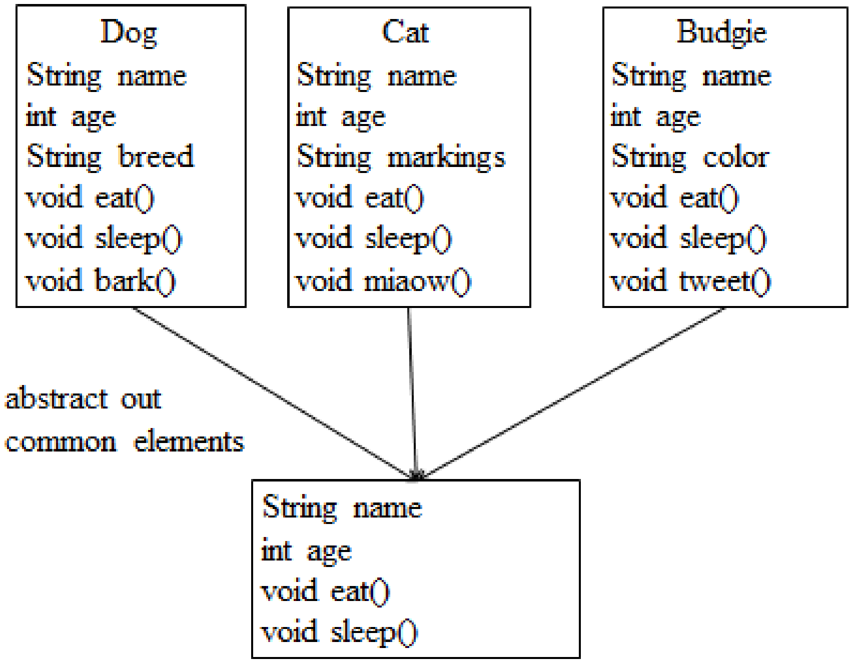
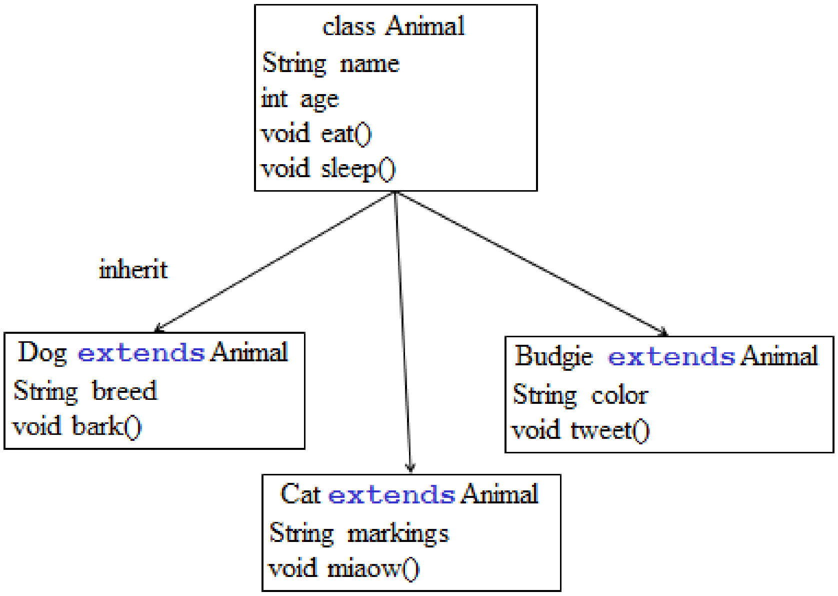
Inheritance

In many situations, we have several classes with common structure – they have instance variables and behaviors in common. In this case, it is appropriate to put these common elements into a single *superclass* and have the separate classes become *subclasses* that *inherit* from it. If a class inherits from another, it has all the instance variables and methods from that class.



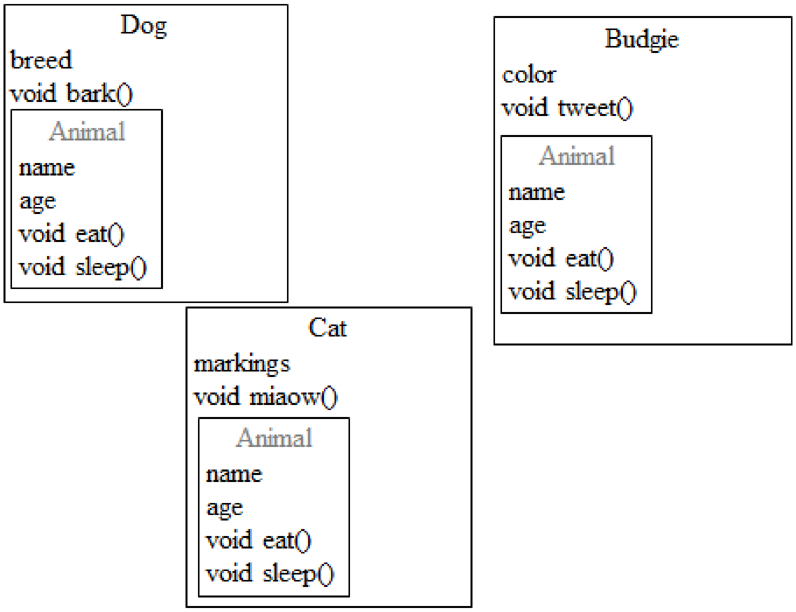
Inheritance represents an *is-a* relationship between classes. We look at the classes involved and identify what they have in common that makes them all special cases of a more general, more abstract category. We *abstract out* what they have in common. For instance we can see that Cats, Dogs, and Budgies all have names and ages and can eat and sleep. These characteristics define the more general type Animal, and we can say Cat is-a Animal, that is Cat is one type of Animal.

The subclasses can add more instance variables and behaviors that are specific to their type.

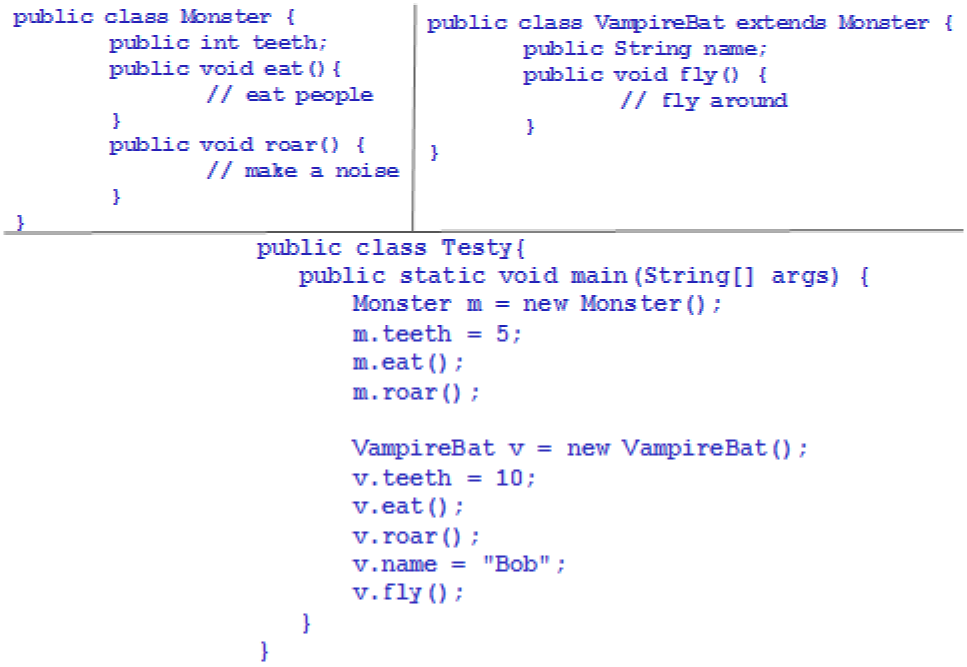


Contrast this with a has-a relationship, in which one type is owned by, or is a component that makes up, another type. A car is-a vehicle. A car has-a motor. For situations best described by is-a we use inheritance. For situations best described by has-a we use *composition* which is the technical term for one class having an instance variable whose type is defined by another class. So, a Car might inherit from type Vehicle, but have an instance variable of type Motor.

If a subclass inherits from a superclass, then whenever we create an instance of that class, an object of the superclass type exists inside the subclass object. In Java this inner object cannot be treated as a separate object or reassigned.



An instance of a subclass can use all the public instance variables and methods it has inherited just as directly, through the dot operator, as any variables or methods directly defined within the class.



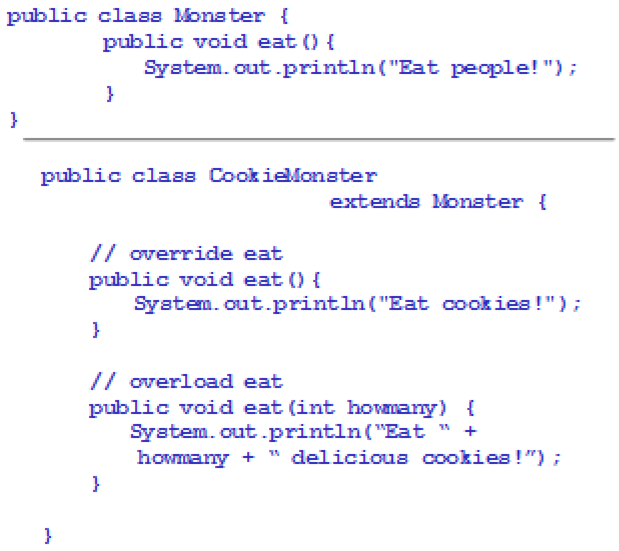
```
public class Monster {
        public int teeth;
        public void eat(){
                // eat people
        }
        public void roar() {
                // make a noise
        }
}
```

```
public class VampireBat extends Monster {
        public String name;
        public void fly() {
                // fly around
        }
}
```

```
public class Testy{
   public static void main(String[] args) {
       Monster m = new Monster();
       m.teeth = 5;
       m.eat();
       m.roar();

       VampireBat v = new VampireBat();
       v.teeth = 10;
       v.eat();
       v.roar();
       v.name = "Bob";
       v.fly();
   }
}
```

Overloading Methods

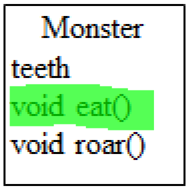
In some cases, an inherited method does not implement the behavior in a way appropriate to the subclass. The subclass can then override the behavior by re-defining the method. (Note that it can *also* overload the method, which is not the same thing.) The notation @Override is used to mark that a method is overloading an inherited method. Doing so acts as a visual reminder that a method replaces an inherited version, but also the IDE checks that a method marked @Override is actually overriding and not accidentally overloading (e.g. by having different parameters).

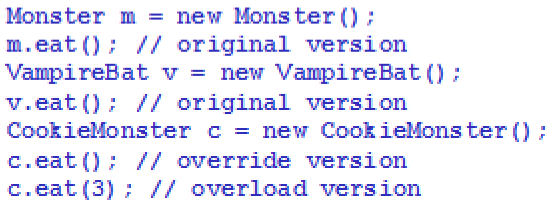


```
public class Monster {
       public void eat(){
          System.out.println("Eat people!");
       }
}
```

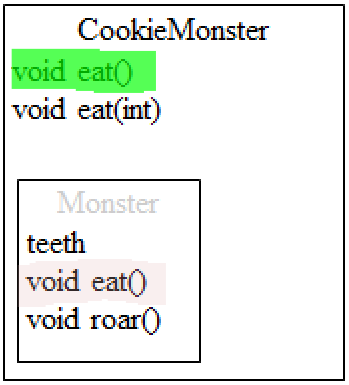
```
  public class CookieMonster
                      extends Monster {

     // override eat
     public void eat(){
        System.out.println("Eat cookies!");
     }

     // overload eat
     public void eat(int howmany) {
        System.out.println(“Eat ” +
        howmany + ” delicious cookies!”);
     }

  }
```

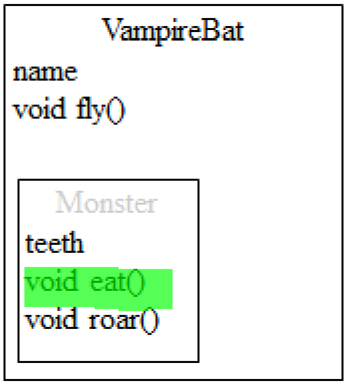
When we call a method for an instance of a class, at compile-time Java simply has to check that the class has that method (either by defining or by inheritance). At run-time, Java looks for the method in the object, and uses the first version it finds, working its way inward – so if the method is overridden, Java automatically calls the version for the subclass, but if not it will find the version in the superclass object within the subclass object.





```
Monster m = new Monster();
m.eat(); // original version
VampireBat v = new VampireBat();
v.eat(); // original version
CookieMonster c = new CookieMonster();
c.eat(); // override version
c.eat(3); // overload version
```





The final keyword can be used on a method to ensure that it cannot be overridden by any subclass, if it is important that all classes do this behavior the same way. If a method is to be called by a constructor (e.g. an accessor or mutator) it is often marked final, so that the constructor can rely on it having specific results.

If the class itself is marked final, then it cannot be inherited from at all.

Hiding Instance Variables

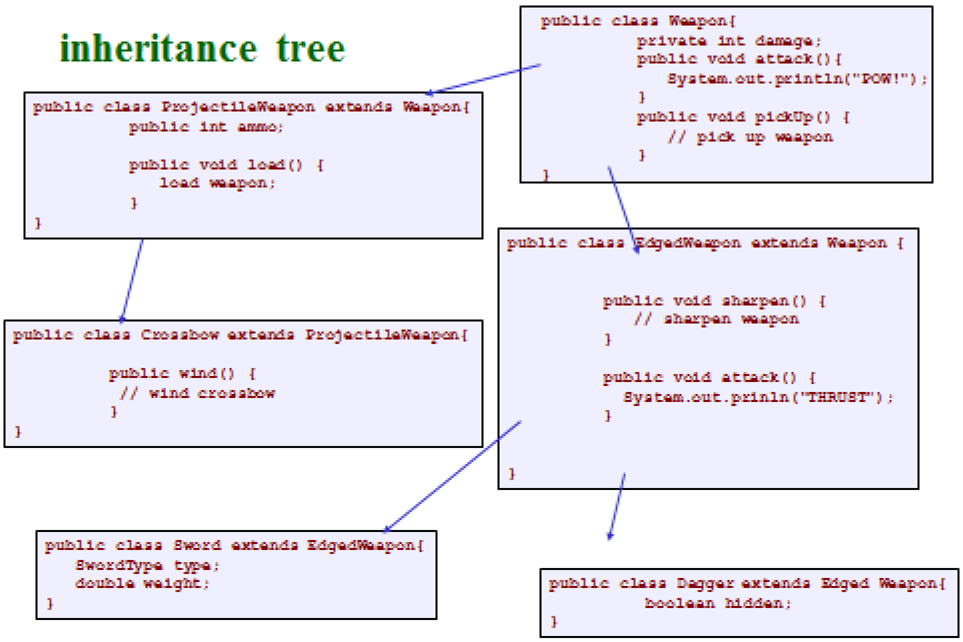
It is legal for a subclass to declare an instance variable with the same name as an instance variable in the superclass. This variable can be exactly the same, or can differ by type, whether the variable is public/private, whether it is final, and whether it is static. In this case, the variable is not *overridden* but *hidden* (the difference won't be clear until we talk about polymorphism).

If we use the variable name in the subclass, it will refer to the subclass version, although we can refer to the superclass version using super. (see later section).

Deliberately naming a subclass instance variable the same as a superclass instance variable is a bad idea. Why is this even legal? Well, suppose it were not. Suppose a subclass has already added an instance variable x, and then you go back and add a variable x to your superclass. If hiding instance variables is illegal, then the subclass is broken!

Inheritance Hierarchy

We can have many levels of inheritance, which we typically imagine as a tree. Nothing is ever inherited up the tree or across from one subclass to another.



```
public class Weapon{
        private int damage;
        public void attack(){
           System.out.println("POW!");
        }
        public void pickUp() {
           // pick up weapon
        }
}
```

```
public class ProjectileWeapon extends Weapon{
        public int ammo;

        public void load() {
           load weapon;
        }
}
```

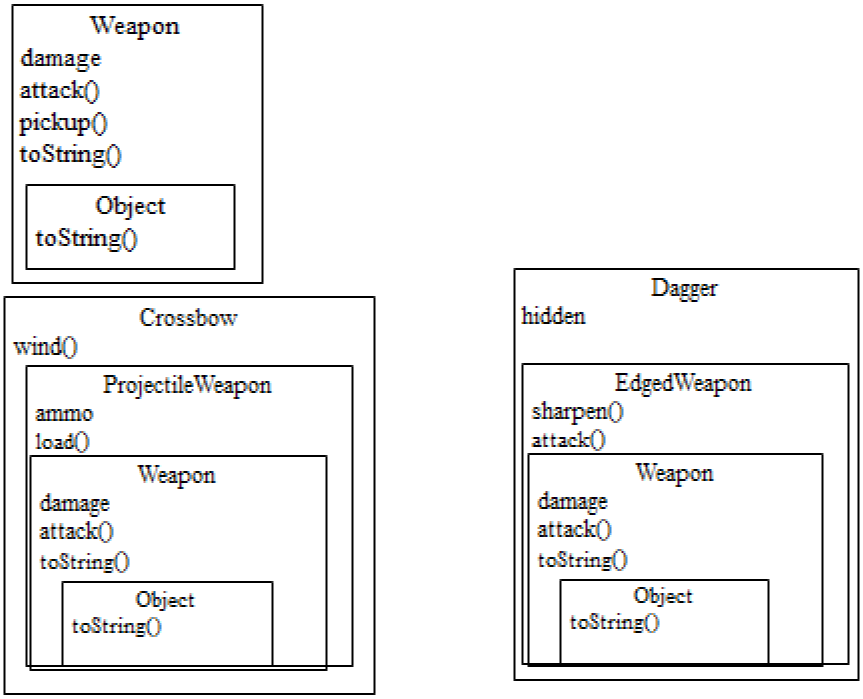
```
public class EdgedWeapon extends Weapon {


        public void sharpen() {
           // sharpen weapon
        }

        public void attack() {
          System.out.prinln("THRUST");
        }


}
```

```
public class Crossbow extends ProjectileWeapon{

        public wind() {
         // wind crossbow
        }
}
```

```
public class Sword extends EdgedWeapon{
   SwordType type;
   double weight;
}
```

```
public class Dagger extends Edged Weapon{
            boolean hidden;
}
```

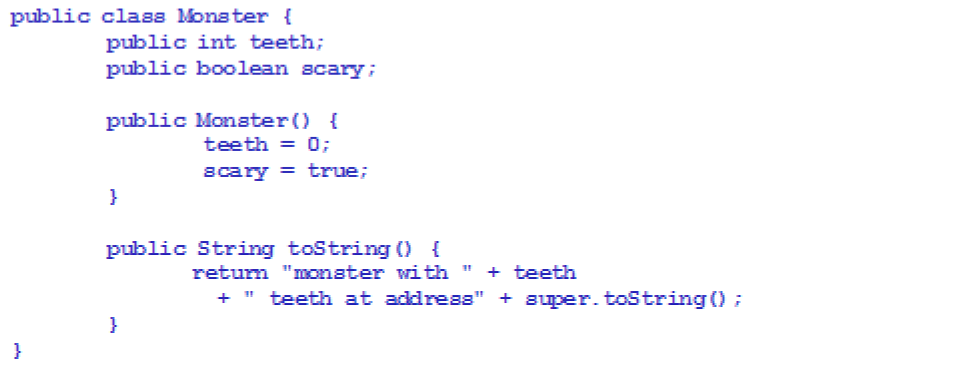
If a class does not explicitly inherit from another class using the keyword extends, then it inherits from the class Object, in which some basic methods such as toString are defined.



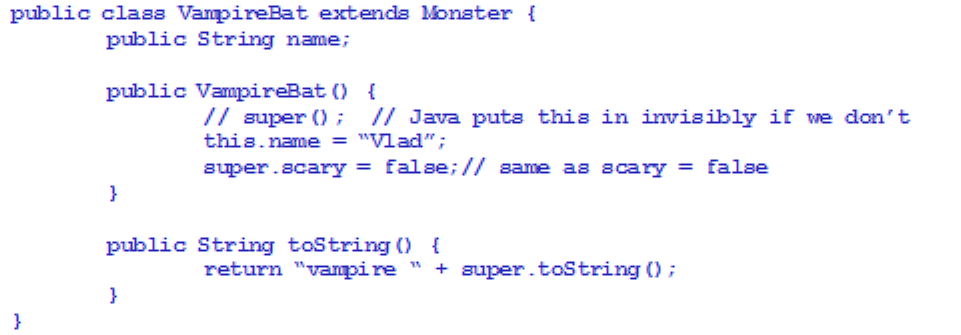
super

The keyword super allows a subclass to refer to its superclass. The super keyword is sometimes used for clarity, like this, to make explicit which variables and methods are inherited and which are specific to the class. It also allows a subclass to call a method from a superclass that it has overridden. This is often used when the subclass wants to do the same as its superclass, but with some additions.

In some ways, super acts like a reference to the superclass object inside the subclass instance, but unlike this, it does not behave like a real reference variable. You cannot, for instance, pass it to a method or assign a variable to its value (or the reverse). You also cannot use super.super or anything similar to work up the inheritance tree more than one level.



```
public class Monster {
        public int teeth;
        public boolean scary;

        public Monster() {
                teeth = 0;
                scary = true;
        }

        public String toString() {
               return "monster with " + teeth
                 + " teeth at address" + super.toString();
        }
}
```

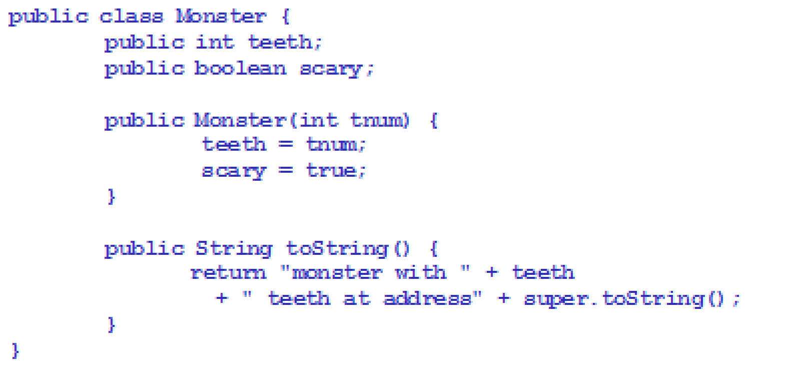


```
public class VampireBat extends Monster {
        public String name;

        public VampireBat() {
                // super();  // Java puts this in invisibly if we don’t
                this.name = “Vlad”;
                super.scary = false;// same as scary = false
        }

        public String toString() {
                return “vampire ” + super.toString();
        }
}
```

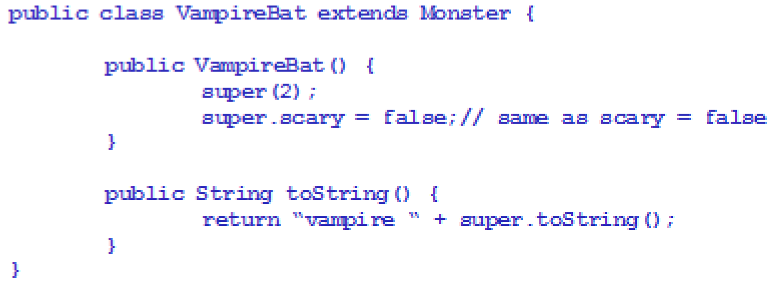
super()

Methods of a superclass are inherited by the subclass, but constructors are not. By default, a subclass constructor silently calls the default superclass constructor as the first line of the subclass constructor.

If you wish to call a parameterized superclass constructor instead, you can do so explicitly using super() (with the appropriate parameters). A call to super() must be the first line of the constructor (so you cannot use both super() and this() in the same constructor). The superclass constructor is called before any initializer block.



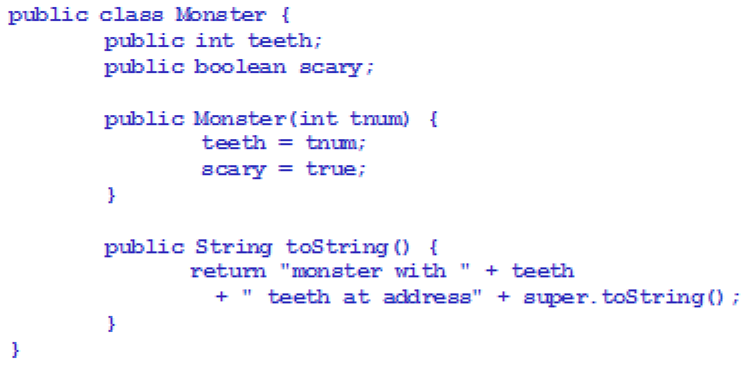
```
public class Monster {
        public int teeth;
        public boolean scary;

        public Monster(int tnum) {
                teeth = tnum;
                scary = true;
        }

        public String toString() {
               return "monster with " + teeth
                 + " teeth at address" + super.toString();
        }
}
```



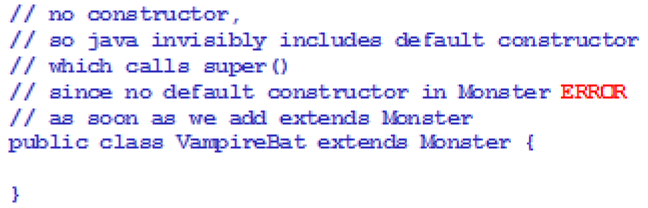
```
public class VampireBat extends Monster {

        public VampireBat() {
                super(2);
                super.scary = false;// same as scary = false
        }

        public String toString() {
                return "vampire " + super.toString();
        }
}
```

As soon as we define a parameterized constructor in a class, Java does not then provide an invisible default constructor as well. So, suppose we do this in the superclass. Then, when we create a subclass, Java will create an invisible default constructor for the subclass, which invisibly calls the default superclass constructor, which doesn't exist. The compiler will then give an error on this invisible line of code in an invisible method as soon as the subclass is marked as extending the superclass.

To fix this, either add a default constructor in the superclass, or a constructor in the subclass that uses super to call the parameterized constructor from the superclass.



```
public class Monster {
        public int teeth;
        public boolean scary;

        public Monster(int tnum) {
                teeth = tnum;
                scary = true;
        }

        public String toString() {
               return "monster with " + teeth
                 + " teeth at address" + super.toString();
        }
}
```



```
// no constructor,
// so java invisibly includes default constructor
// which calls super()
// since no default constructor in Monster ERROR
// as soon as we add extends Monster
public class VampireBat extends Monster {

}
```

public, private, protected

Elements of a class that are public are available to all other classes.

Elements of a class that are private are not available to any other classes, including subclasses. This means that subclasses cannot access their own inherited instance variables (except through accessors and mutators).

The keyword protected marks an access level between public and private. If a variable is protected, it is directly accessible to subclasses as well as the class itself. However, protected variables are *also* available to all classes in the same package. We have been using single packages for simplicity. Remember not to misuse this to get direct access to instance variables from a test harness, for example.

protected access simplifies writing code, however, like making parts of a class public, it locks us to one implementation. We cannot remove or change the type or name of any protected variable (or method) without potentially breaking subclasses. So, it is generally safer coding to have the subclasses go through accessors and mutators.

Inheritance and Static

If a class has a static variable, there is only one copy of that variable for that class and all its subclasses. It can be accessed with the dot operator using the name of the superclass or any subclass. The same goes for static methods.

It is not possible to override a static method, but it is possible to hide it (again, the distinction involves polymorphism). So, it is legal to define a static method in a subclass which has the same signature as a static method in the superclass.