Inheritance and Polymorphism

Inheritance

When we design the classes for the object types in a program, there can be various relationships between them that we want to represent. We have seen one kind of relationship, with one class having an instance variable whose type is another class, like a Dog having a buddy that is a Cat. But another kind of relationship is that one class is a more specific version of another class, or multiple classes are different versions of something more general. To represent this kind of relationship, we use *inheritance* between the classes.

Suppose in our design we realize that Dogs and Cats and Budgies all have a lot of things in common: they all have a name and an age, and all can eat and sleep and do these things in roughly the same ways

Dog
String name int age String breed Cat buddy
void eat() void sleep() void bark()

Cat
String name int age
void eat() void sleep() void miaow() void eat(Budgie*)

Budgie
String name int age
void eat() void sleep() void tweet() void fly()

Thinking about dogs, cats, and budgies, they don't just have a lot of overlapping elements, they are conceptually related too: they are all kinds of living things, kinds of animals, kinds of pets.

When we recognize that the components of the situation we are designing classes for have this kind of overlap, we *abstract out the common elements*. This means that we take the common elements and try to identify what type of object that area of overlap would represent. This will be something more abstract – more general and less specific – than the classes we started with.

Note that if we look at the behaviors we identified for the three animal types, there's another thing other than eat and sleep that they have in common. Dogs bark, Cats miaow, and Budgies tweet, and these are all ways of making noise. In the process of abstracting

out, we might decide to simplify our design and just have one method name for all of these, calling it makeNoise().

Since these animal classes all have names, let's decide that this more general class should be Pet.

Pet
String name int age
void eat() void sleep() void makeNoise()

So now instead of three classes with overlap, we will create first one Pet class to contain the common elements, and then create the other three classes, each of which *inherit* from the Pet class, which means they will get everything that Pet has, without having to actually copy the code. This means we only have to write the common elements once, and if we need to update them, we only have to update one place, but all the inheriting classes will use those updates. Each of those classes can also add anything it needs that makes it different from the original Pet.

The keyword `extends` is how Java indicates inheritance. Even when inheriting, remember that every Java class lives in its own file, named the same as the class.

```
// a normal class, just as we've been creating
public class Pet {
    public String name; // only public for example purposes
    public int age;     //   we know better!

   // …accessors, mutators constructors, toString here

    public void eat() {
        System.out.println(name + " eats some food");
    }

    public void sleep() {
        System.out.println(name + " sleeps.  Honk-shu, honk-
   shu");
    }
```

```
    public void makeNoise() {
        System.out.println(name + " says eeeep");
    }
}
```

```
// Dog inherits everything from Pet
// and adds an instance variable
public class Dog extends Pet {
    public String breed;     // oh no, public again!
    public Cat buddy;        // just for the example

   // needs accessors, mutators, constructors, toString

}
```

```
// Cat inherits everything from Pet
// and adds a method
public class Cat extends Pet {
    public void eat(Budgie victim) {
         // the Budgie has a name, it was inherited
        System.out.println(name + " eats " + victim.name);
        victim.name = "dinner";
    }

    // needs constructors and toString

}
```

```
// Budgie inherits everything from Pet
// and adds a method
public class Budgie extends Pet {
    public void fly() {
         // the Budgie has a name, it was inherited
        System.out.println(name + " flaps into the sky");
    }

    // needs constructors and toString
```

```
}
```

```
// in a separate main class
public static void main(String[] args) {
    // p has all the things in the Pet class
    Pet p;
    p = new Pet();
    p.name = "generic pet"; // accessible because public - yuck!
    p.age = 0;
    p.eat();
    p.sleep();
    p.makeNoise();

    // b has all the things in the Pet class and Budgie class
    Budgie b;
    b = new Budgie();
    b.name = "Tweety";  // inherited
    b.age = 1;          // inherited
    b.eat();            // inherited
    b.sleep();          // inherited
    b.makeNoise();      // inherited
    b.fly();

    // c has all the things in the Pet class and the Cat class
    Cat c;
    c = new Cat();
    c.name = "Kitty McFreckles";  // inherited
    c.age = 6;                    // inherited
    c.eat();                      // inherited
    c.sleep();                    // inherited
    c.makeNoise();                // inherited
    c.eat(b);

    // d has all the things in the Pet class and the Dog class
    Dog d;
    d = new Dog();
    d.name = "Spot the Dog"; // inherited
```

```
        d.age = 3;                  // inherited
        d.eat();                    // inherited
        d.sleep();                  // inherited
        d.makeNoise();              // inherited
        d.breed = "Hungarian Hamburger Hound";
        d.buddy = c;

        // error: no access to things in other classes we did
        // not inherit from
        b.breed = "flying dogbird?"; // Budgies don't have breed
        c.fly(); // Cats don't have fly
        d.eat(b); // dogs don't have eat that takes a Budgie
    } // end main
```

When we have inheritance in our class design, we say that the class inherited from is the ***superclass*** or *parent class* and the classes that inherit from it are the ***subclasses*** or *child classes*. Each subclass inherits (almost) everything that was in the superclass, but notice that they do not inherit everything in every related class – Dog isn't inheriting what Cat has, only what Pet has.

Although we casually would say that "Dog inherits everything from Pet" actually *constructors are not inherited*, and anything marked static is not inherited (so another class couldn't inherit your main). We will come back to this issue of constructors for subclasses when we talk about super().

protected

In these examples I made instance variables public so we could see that the subclass objects really do have those variables. We know that isn't correct.

We have been making our instance variables private, and that's usually still the right answer. But note that if name was private in Pet, Dog would have to access *its own name* through the accessor and mutator just like any other class would. That's not a terrible idea; we often have a class try to go through its own accessor and mutator to guarantee validation and to future-proof against changes.

We do sometimes instead use a third choice: protected. Protected makes instance variables directly available to subclasses, which is sometimes very convenient – if my

subclass is inheriting an array instance variable, we should probably give it full access to the array.

There is a little problem however: protected also makes instance variables directly available to other classes in the same package. We have always just stuffed all our classes into one big package, but in a real java program we would divide up into packages to help organize our code; the class with main would usually not be in the same package with the classes it uses. So technically, any class, including the one with main, would be able to directly access any protected instance variables, because we are being sloppy about packages.

When you use protected, it is your responsibility to remember not to have main or other classes violate the basic concepts of encapsulation in Java. That is: never have code like main directly access instance vars instead of going through the accessor or mutator. If you can't trust yourself, use private instead.

Has-A and IS-A Relationships

When we are designing the classes for our program, we may have figured out that two classes have a relationship but we're not sure whether it is better to represent that relationship through inheritance, or just having an instance variable. We use the terms *IS-A* and *HAS-A* to help identify which it should be: if we would describe the relationship as "is a", then it should be inheritance; if we would describe it as "has a", it should be an instance variable.

A dog has a cat buddy – HAS-A: Dog should have an instance variable for a Cat

A dog is a type of pet – IS-A: Dog should inherit from Pet.

Sometimes when programmers are learning to program with object orientation, they are tempted to create lots of inheritance relationships just because they want to access variables or methods from one class inside another. Think through HAS-A vs IS-A before doing this, and if the relationship is HAS-A, use an instance variable, not inheritance.

Part of the point of object orientation is to make a complex program easy to understand, so we should only have inheritance between classes if that correctly represents our understanding of the relationship between them, that one is a more specific version of the other. If the way the relationship between the classes is written isn't helping us understand the program, then our object oriented design isn't doing a good job.

Multiple Levels of Inheritance

So far we have only had a superclass and some subclasses, but often the relationships between the classes identified in our overall program design involve multiple layers, and a class that is a subclass of one class needs to also be a superclass to another.

If we add a class FloatingCat that is a subclass of Cat, which is itself a subclass of Pet, it will inherit everything from Cat, which includes everything from Pet.

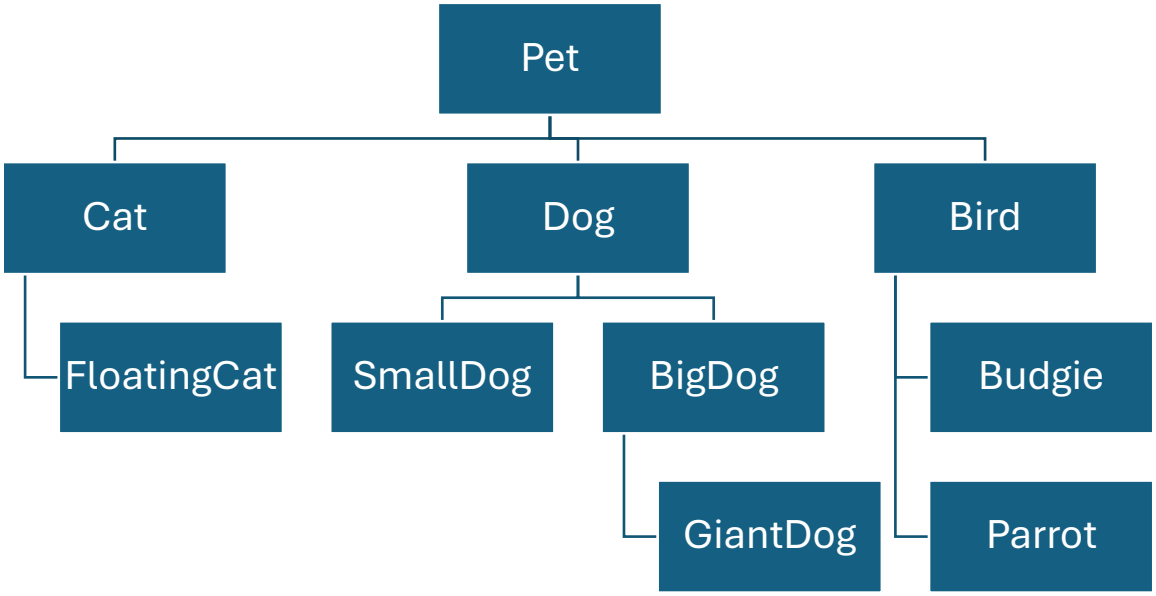
```
// Floating Cat inherits everything from Cat
// which includes everything from Pet
// and adds an instance var
public class FloatingCat extends Cat {
    // how far off the floor this cat floats
    public double floatingHeight; // public again, oh no!
} // end Cat
```

```
// in another class with main
public static void main(String[] args) {
    // b has all the things in the Pet class and Budgie class
    Budgie b;
    b = new Budgie();
    b.name = "Tweety";  //inherited
    // c has all the things in the Pet class and the Cat class
    Cat c;
    c = new Cat();
    c.name = "Kitty McFreckles";  //inherited
    c.age = 6;                    //inherited
    c.eat();                      //inherited
    c.sleep();                    //inherited
    c.makeNoise();                //inherited
    c.eat(b);
    // f has all the things in the Pet class and the Cat class
    // and the FloatingCat class
    FloatingCat f;
    f = new FloatingCat();
    f.name = "Koshekh";      //inherited
    f.age = 90008;           //inherited
    f.eat();                 //inherited
    f.sleep();               //inherited
```

```
        f.makeNoise();              //inherited
        f.eat(b);                   //inherited (from Cat)
        f.floatingHeight = 4.3;
    } // end main
```

Having multiple levels of inheritance allows us to represent more kinds of relationship and have finer grained control of which classes share variables and methods. We often picture inheritance as a tree (in computer science, trees have branches going downward).

We might eventually revise our design to reflect that all dogs have things in common, but there are also differences between big dogs and small dogs. We might also decide that giant dogs are like big dogs, except that they can be ridden by babies. We might also decide that we want to have a Parrot as a valid type for pets, and notice that they have enough in common with our existing Budgie class that we add a new class Bird as a superclass of both, and have that inherit from Pet (indicating that in this context, all the birds that matter to our program are pets).



Overriding

In our earlier example, we wrote the methods eat(), sleep(), and makeNoise() in Pet and then the other classes inherited them. This meant that when we tell a Dog, Cat, or Budgie to eat or sleep or make noise, they all do it exactly the same way. But often a subclass will need to change an inherited behavior method to correctly reflect the class design.

In that case we will re-write the body of that inherited method in the subclass, replacing the version from the superclass. This is called *overriding*.

Let's have Dog override the makeNoise method; going "eeeep" is close enough to the noise our Cats and Budgies make, but we want Dogs to say "ruff".

```
// Dog inherits everything from Pet
// and adds an instance variable
// and overrides makeNoise()
public class Dog extends Pet {
    public String breed;
    public Cat buddy;

   // overriding keeps exactly the same method header
   // but changes the body of the method
    public void makeNoise() {
        System.out.println(name + " says ruff!");
    }
} // end Dog
```

Now, when Cats or Birds are told to make noise, they use the inherited version from Pet, but when Dogs are told to make noise, they use the Dog-specific version.

If we have subclasses of Dog like SmallDog and BigDog and GiantDog, they will all inherit the overriding version from Dog and say "ruff" although any of them could also override that, so we could decide that small dog's say "yip" instead by adding another override of makeNoise in the SmallDog class.

In addition to inheriting eat() from Pet, Cat had a special eat(Budgie) method just for eating Budgies; remember that having two methods with the same name but different parameters is called overloading – this is different from overriding – Cats have two eat methods, the inherited one from Pet, and the overloaded version

```
public static void main(String[] args) {
    // p has all the things in the Pet class
    Pet p;
    p = new Pet();
    p.name = "generic pet";
    p.eat();
    p.makeNoise(); // "generic pet says eeeep"
```

```
    Budgie b;
    b = new Budgie();
    b.name = "Tweety";
    b.makeNoise(); // “Tweety says eeeep”

    Cat c;
    c = new Cat();
    c.name = "Kitty McFreckles";
    c.makeNoise();  // “Kitty McFreckles says eeeep”

    Dog d;
    d = new Dog();
    d.name = "Spot the Dog";
    d.makeNoise(); // “Spot the Dog says ruff!”
} // end main
```

In these examples I have not been showing accessors, mutators, or toString. Instance variables still need these[1] when we are doing inheritance. Accessors and mutators should be created in the class that creates the instance variable, but subclasses might override them, for instance a subclass might add extra validation to the mutator if it has extra requirements about what values an instance var can have.

Most classes need their own override of toString so that they can be printed with meaningful output. Subclasses that add instance variables *definitely* need their own override of toString that adds their variables.

When you override a method in Netbeans, it will suggest that you add the annotation @Override above the method. This is not required to get overriding to work, but it is a convenient tool. Suppose that later you forget that a certain method was an override, and you go back and change its name or parameters. This would make it no longer an override, just a totally unrelated method, and any code using this class would suddenly be calling the original inherited version of that method. @Override tells java to check for this and if a method marked as an override no longer matches an inherited method, it will show an error.

---

[1] Even if they are protected? *YES. Of course.* Protected is about convenience for subclasses, but we still want to keep all the advantages we got from encapsulation. Protected elements are still in the private interface from the point of view of non-subclasses.

It will turn out that overriding is not a rare occurrence. Being able to override allows us to have a set of shared behavior methods among many classes, but still give each one the flexibility to adjust how they do any of them. It also sets us up for the biggest concept in object oriented coding: Polymorphism.

Object Inheritance

We mentioned earlier that Netbeans encourages you to use the @Override notation on overriding methods. But there's a method that Netbeans has been encouraging you to use this on since we first learned about classes – it wants to put @Override on toString. Which means that toString must have already been an override. Which means we must have inherited a version of toString… but from where?

In Java, if a class is not marked with the extends keyword as inheriting from some other class, it automatically inherits from a class called Object. The Object class is a very simple class that just has a couple very general methods like toString. The toString in Object is pretty bad – after all, they didn't know what our class was going to be or what instance variables it would have -- which is why we always override it if we have any instance variables.

Object, then, is always the top of any inheritance tree in Java, and every class we wrote before we got to inheritance was already inheriting from Object.

super. and super()

We mentioned earlier that constructors are not inherited by subclasses; each subclass must write its own. But we do want subclasses to be able to have their superclass do the work of setting up its own instance variables. So we have a way of chaining subclass constructors back to superclass constructors. It turns out we can also chain from an overriding method to the method it replaces. All of this is based on the keyword `super`.

`super` is very similar to `this` in that it is used both for special syntax in the constructor and can be put in front of the dot operator for readability. Chaining other methods is something `super.` can do that `this.` cannot, but also although we have `super()` and `super.` like `this()` and `this.`, we do not have just `super` as we do have just `this`

super() for the default constructor

super() is how a subclass constructor calls a superclass constructor. This follows the same rule as for this() – it can only be done as the first line of a constructor. But if you do

just want your subclass constructor to call the default superclass constructor, you don't have to write that code, a call to super() is included for free.

```
public class Pet {
  public String name;
  public int age;

   // NOT inherited
   // is super() from Dog's point of view
   public Pet() {
         name = "pet";
         age = 1;
   }
```

```
public class Dog extends Pet {
   public String breed;

   public Dog() {
     // super();  // Java puts this in invisibly if we don't
         // so Dog's name starts as "pet" and age as 1
     super.name = "Spot";// same as   name = "Spot
     this.breed = "mutt";// same as   breed = "mutt"
   }
}
```

So in the above code, even though the line in Dog() that calls Pet() using super() is commented out, it happens anyway, and the dog's name and age both start with the default values from Pet.

super. for readability

Notice that in Dog when I wanted to change the name, I called it `super.name`, but called the breed `this.breed`. Remember that we often put `this.` in front of instance variables to make it more visible that they are instance vars instead of locals. Putting `super.` in front of an inherited instance variable is a way of emphasizing that it is inherited. Just like `this.`, `super.` doesn't change how the code works, it's just for readability. Note that since name was inherited and is owned by the Dog class, in the Dog class we could also say `this.name` if we wanted to; that is equally valid. We could not say super.breed though, since breed is not inherited.

super(...) for parameterized constructors

If all you need is the default constructor from your superclass, you never have to actually write the code to call `super()`. But suppose that you want your subclass constructor to take advantage of a parameterized constructor in your superclass. Then we do need to explicitly use `super(… )` but with parameters in the parentheses.

```
public class Pet {
  public String name;
  public int age

    // super()
   public Pet() {
         name = “pet”;
         age = 1;
   }
    // super(name, age)
    public Pet(String n, int a) {
         this();
         name = n;
         age = a;
   }
```

```
public class Dog extends Pet {
   public String breed;

   public Dog() {
     super(“Spot”, 3); // call Pet’s parameterized constructor
     this.breed = “mutt”;// same as breed = “mutt”
   }
}
```

In this example, Pet has a parameterized constructor that takes values for the age and the name. To call this from the constructor in Dog, we user super(...) and put the default name and age for Dogs into the parentheses to pass those values to Pet’s parameterized constructor.

Watch out for missing default constructors in superclass!

Suppose we wrote all of Pet before starting Dog, and in Pet we had written the parameterized constructor, but not the default constructor. When we then went to write Dog, we would get an instant error as soon as we added "extends Pet"!

To understand why, we have to put a couple things together:

- If you wrote no constructors in a class, Java puts in an invisible default constructor that just does nothing. That's why we were always able to use new on our classes even before we knew about writing constructors.
- However, as soon as you write one constructor, Java no longer puts in the invisible default.
- If you don't specify that you are calling a parameterized superclass constructor in your constructor, Java puts in an invisible line of code to call super()

If we add this up: if Pet has a parameterized constructor but we didn't write a default, then Pet does not have a default constructor. If Dog has no constructors yet written, then Java puts in the invisible default, and in that invisible default, it puts in the invisible line that calls super(), the default superclass constructor… which doesn't exist. So we are getting an error on an invisible line of code in a method we can't see.

If you are in this situation, there are two possible fixes. Most of the time, the right answer is to go back and put a default constructor in your superclass. But it can be correct for some classes to not have have a default constructor: if it only makes sense to create an instance of that class when given values up front and creating a default would be invalid. In those cases, you'll just have to wait until you're done writing the subclass constructor that calls the correct parameterized superclass constructor for this odd error to go away.

Using super. to chain other methods when overriding

Just as we can chain from a subclass constructor to a superclass constructor using `super.`, we sometimes want to chain from a subclass method that overrides something from the superclass to that overridden method.

We often see this in toString: suppose the superclass has written a very nice toString for all of its instance variables, and the subclass wants to use that same format for those, but needs to add something of its own. Then we can use `super.toString()` in our subclass toString so that we can include everything from the superclass version without rewriting that code.

```
public class Pet {
    public String name;
```

```
    public int age

    public String toString() {
         return name + "(" + age + " years) “
               + super.toString();
    }
```

```
public class Dog extends Pet {
    public String breed;

    public String toString() {
          return super.toString() + “ a “ + breed;
    }
}
```

In this example, Pet's toString is chaining to the toString that it inherited from the Object class (we usually wouldn't do that; the toString in Object is mostly useless, but I wanted to show that it can be a chain of many links). Then the toString in Dog is chaining back to the one in Pet, including everything it has, but putting "a" and the breed after it.

Chaining when we override instead of re-writing the code completely is often a smart thing to do because it means that if the superclass updates or adds its code we will automatically include those changes. But this only works if we want our method to add its own steps before and after the superclass version. If we want to replace the superclass version, or insert something in the middle, we'll have to override without chaining.

Polymorphism

Depending on the actual program, we could have ended up with a lot more in our inheritance tree for pets: we could add branches to the inheritance tree for reptile pets, and rodent pets, and insect pets, and fish pets. When we then go to write our main code, we will probably have some parts where we are only using the methods they all have in common (the inherited methods). In that case, it is a bit annoying that we would have to re-write the code multiple times for different classes, when the only difference is the type of the variable.

In fact, we can write the code just once, using type Pet, and it will work for all the subclasses!

Polymorphism means that a variable of a superclass type can actually hold the address of any subclass type. It can only use variables and methods that are legal for the variable

type, not those that the subclass may have added. However, when the program runs, any methods called will always be the correct version if the subclass overrides them.

```
//in a main
    // Pet variable but pointing at a Dog
    Pet p = new Dog();
    p.name = "Fluffy";
    p.sleep(); // original sleep method
    // since p is actually a Dog, call override method
    p.makeNoise(); // “Fluffy says ruff!”

    // error, pets don’t have a breed
    p.breed = "nope";

    // if we want to do dog-specific code,
    // we need a Dog variable
    Dog d = new Dog();
    d.breed = "Beagle";
```

Note that when I tried to use the breed instance variable for the Pet variable p, that was an error, even thought p was actually holding the memory address of a Dog, which does have breed. What is legal to put after the dot operator is checked at compile time based on the type of the variable, not the type of the object!

If we wanted to do Dog-specific code, we needed to create a Dog type variable. Polymorphism isn’t appropriate to use when we want to write code that is specific to one subclass.

However, polymorphism does give us a way to write code that will work for any subclass, and still get different results for different classes, based on overriding

```
    Pet p;
    System.out.println("Let’s try a starter pet!");
    System.out.println("dog, cat, or budgie?");
    String user = scan.nextLine();
    // depending on user input, Pet-type variable
    // gets pointed at Dog, Cat, or Budgie
    if (user.equals("dog")) {
        p = new Dog();
    } else if (user.equals("cat")) {
        p = new Cat();
```

```
    } else {
        p = new Budgie();
    }

    // this code works no matter which type they chose
    System.out.println("what name?");
    p.name = scan.nextLine();
    System.out.println("how old?");
    p.age = scan.nextInt();
    p.eat();
    p.sleep();
    p.makeNoise(); // method called depends on user choice
```

Now that we have polymorphism, we can make Dog a little more flexible about being friends; instead of buddy being limited to Cats, we could make it a Pet

```
public class Dog extends Pet {
    public String breed;
    public Pet buddy;

    public void makeNoise() {
        System.out.println(name + " says ruff!");
    }
}
```

```
// in a separate main
    Dog fluffy;
    fluffy = new Dog();
    fluffy.name = "Fluffywuffykins";

    System.out.println("Choose a friend for " + fluffy.name);
    System.out.println("dog, cat, or budgie?");
    String user = scan.nextLine();
    if (user.equals("dog")) {
        fluffy.buddy = new Dog();
    } else if (user.equals("cat")) {
        fluffy.buddy = new Cat();
    } else {
        fluffy.buddy = new Budgie();
    }
```

```
// this code works no matter which type they chose
System.out.println("what name?");
fluffy.buddy.name = scan.nextLine();
System.out.println("Now they will talk");
fluffy.makeNoise(); // dog version
fluffy.buddy.makeNoise(); // method depends on user choice
```

Also note that an array of pointers to a superclass type can hold any subclass types

```
Pet[] petList = new Pet[10];
// fill the array with various types of pet
for (int i = 0; i < petList.length; i++) {
    if (i % 3 == 0) {
        petList[i] = new Dog();
    } else if (i % 3 == 1) {
        petList[i] = new Cat();
    } else {
        petList[i] = new Budgie();
    }
    // starter name
    petList[i].name = "Pet #" + (i + 1);
}

// this code works for all pets
for (int i = 0; i < petList.length; i++) {
    petList[i].eat();
    petList[i].sleep();
    petList[i].makeNoise(); // outcome depends on i
}
```

If we want to set values for instance variables specific to the subclasses, we can't do that after they're in the array because then we're locked into only things that Pet has. But this is something we could easily handle by giving each subclass a nice parameterized constructor.

Note how simple the loop at the end of this example is. There's no if to check whether we should be doing a special different makeNoise method for Dogs, it just happens automatically. If we did more overriding of the Pet methods in different classes, the outcome of that loop could be totally different for each type of pet, but this polymorphic

code in main doesn't have to change at all to reflect this. The part of the programming team writing the main code doesn't even have to know whether any of the subclasses overrode a method from the superclass or not.

This is how real sophisticated Java programs can be written and maintained: once we agree on the design of the superclass, one team can write polymorphic main code that will work for any subclass, but they don't have to know which of those subclasses will override what. Each subclass team can decide what overrides are a good idea for making their subclass work the way it should, but they don't have to worry about anybody else's classes. Even years later, someone can go back into the code of one subclass and add an override to change how something works, but this won't require that the main code be changed, it will just work differently.

Polymorphism and overloaded methods

Remember that we can write two versions of a method with the same name in the same space if we give it different parameters; this is over*loading* not over*riding*. What if the different parameter types are linked by inheritance? Once again, what happens will be based on the variable type at compile time, not the actual object type at runtime, so if we do

```
//overload takes a Pet
public static void petStuff(Pet p) {
   p.eat();
   p.sleep();
}

// overload takes a Budgie
public static void petStuff(Budgie b) {
   b.fly();
   b.eat();
   b.sleep();
}

//elsewhere...
   Pet p1 = new Budgie();
   petStuff(p1); // calls Pet version
   Budgie p2 = new Budgie();
   petStuff(p2); // calls Budgie version
```

Both the p1 and p2 variables hold the memory addresses of Budgies, but one is stored in a Pet variable and the second in a Budgie variable. When we call the petStuff method, p1 will be sent to the petStuff version that takes a Pet, not the Budgie version, because this is determined by variable type

equals

In most cases, we are only writing polymorphic code when we want polymorphic behavior, so it is fine not to be able to tell what actual subclass object is stored in a superclass variable, but there are situations where we have to be in a polymorphic situation but we actually need to deal with subclass specifics.

An example of this is the equals method.

We have so far only seen this for Strings. We know that we have to use `.equals()` for Strings instead of == because == just compares memory addresses, not the letters in the String.

Well, for our own classes we have a similar problem: == would only compare two Dogs or Pets for the same memory address, but what if we want two Dogs to be considered equal to each other if they have the same name, age, and breed? In that case we have to write our own .equals method. This will be an override, since we did inherit an .equals from Object, but that one just does the same thing as ==.

Because .equals is an override, we must match the header of the method from the class Object, which takes an Object

```
public boolean equals (Object obj) {
```

Beginning Java programmers are often tempted to write .equals that takes the type they're in, for instance having the .equals in the Pet class take a Pet parameter instead of an Object parameter. *Do. not. do. this*[2].

It isn't technically a syntax error to write a .equals that takes a different type, but it will break things because you will in that case be overloading rather than overriding, so your class will technically have *two* .equals methods, and eventually a situation will come up where the wrong one will get called. *Java programmers do not overload .equals, they override .equals that takes Object*.

[2] I understand that it looks to you like it should work. I understand that if you write enough versions of the .equals method, it will *seem* like it works in all the examples you can't think of to test. It doesn't work in the long term. A surprising amount of Java library code relies on you doing this correctly.

The equals method should return true if `this` is the same as the Object passed in, false otherwise. By polymorphism, the Object parameter (which I am calling obj) can hold an instance of any class, since everything is a subclass of Object. So if we are checking for equality of Pets, obj may really be a Pet. But it could also be a Book or a Scanner, so how can we find out what it really is, and if it is a Pet, how do we get Java to treat is as one so we can check the right instance variables?

To check types, we can use the `instanceof` operator or the getClass() method inherited from Object.

Note that instanceof is not camelCased, it is not a method, it is an operator like == or >, and is used like this:

```
obj instanceof Pet
```

results in a boolean that is true if the variable obj is holding the memory address of an instance of the Pet class or of any subclass of Pet. If obj is holding a null, this is automatically false.

The getClass() method is an example of reflection in Java, the idea of a Java object having knowledge of its own type. We could either compare to a specific class

```
obj.getClass() == Pet.class
```

or we could compare to the class of the current object

```
obj.getClass() == this.getClass
```

Note that in either case this is an exact match; it doesn't match subclasses like instanceof does. Which of these we want depends on how we want equality to work. Note that since getClass is a method called with the dot operator, we have to check for null before this is safe to do.

However we do the check, we're still not done. Remember that what we're allowed to do depends on the variable type. As long as our variable is type Object, we can't look at any of the Pet-specific variables even though we know it is actually a Pet.

But we do already know how to tell Java to treat a value as a different type – we cast it by putting the type in front of the value in parentheses. That allows us to move the Pet object from the Object variable into a Pet variable so we can use it as a Pet:

```
Pet temp = (Pet)obj; // cast obj to type Pet
```

Let's do an example equals method using instanceof

```
public class Pet {
  private String name;
  private int age;

@Override
  public boolean equals (Object obj) {

   // only other Pets could be equal
   if (obj instanceof Pet) {

     // cast so we can see class specific info
    Pet pobj = (Pet)obj;
     // equal if name and age are same
     return pobj.getAge() == this.getAge() &&
           pobj.getName().equals(this.getName());
   }

   // if any of that failed, they are not equal
   return false;
  }
}
```

In this example, I decided that for two Pets to be equals to each other, they had to have the same name and age. I used == for age, since that's an int. I used .equals for Strings, as we always have – Pet's .equals is using String's .equals as a tool. Almost classes in real java code have .equals, just as they have toString.

So in general, your subclass should also have an equals. It can usually chain back to the superclass equals using `super.` , for instance, in Dog:

```
public class Dog extends Pet {
  private String breed

  @Override
  public boolean equals (Object obj) {
   // let superclass check Pet stuff
   // then check it is the right type
   if (super.equals(obj) && obj instanceof Dog) {
```

```
     // cast and check Dog-specific variable
     Dog dobj = (Dog)obj;
     return getBreed().equals(dobj.getBreed());
    }

    // if any of that failed, it is not equal
    return false;
   }
 }
```

In this subclass .equals method, we used super.equals to chain to the Pet's .equals and let it handle checking the age and the name. This way if the superclass later adds more instance variables, it can handle checking them and we'd get that for free by chaining to its equals method. But since Dog has more instance variables, we also needed to check that it was actually a Dog; if so, we cast it, and then we were able to compare breeds.

Note that we used instanceof, which matches subclasses. This means that an object of type Pet can be equals to an object of type Dog. Depending on our program, that may or may not make sense.

If we used `(obj.getClass() == this.getClass())` in the Pet version of equals we could still chain, but a Pet could not be equals to a Dog because getClass with == doesn't include subclasses[3]. In some situations, it is important that a superclass object never be considered equal to a subclass object.

If we used `(obj.getClass() == Pet.class)` in the Pet version of equals, this chaining wouldn't work, and we would be saying that our equals in Pet only makes sense for Pets, and each subclass needs to write its own equals from scratch.

The first time programmers see tools like getClass() or instanceof, they tend to get tempted to use them whenever they realize they need something specific from a subclass in writing polymorphic code. Remember that we needed them for equals because we were stuck in a polymorphic situation but we didn't really want to be writing polymorphic code; we wanted our code to be specific to our own classes. In general, it is much better to go back into our inheritance tree structure and override methods to get different behaviors for subclasses rather than using instanceof. So, consider instanceof your last resort when you can't find a

[3] Make sure you understand why chaining still works. If I (this) am a Dog running my own .equals, and that chains to the Pet version of .equals, it is still me (this) the Dog who is running super.equals so my class (this.getClass() is still Dog)!

clean polymorphic was to write the code, or else consider it a sign that some code should just be specific to the class type, not polymorphic at all.