Objects

Most languages give us basic types to store numbers, text, and booleans, and arrays to store lists of values. Object Oriented languages provide tools for creating user-defined types that combine storage for multiple types together, alongside methods, to represent more complex components of a program.

The *Procedural Paradigm* in programming design thinks of a program as a series of steps. This series can be very complex, making decisions and looping to repeat , but ultimately it can only design a program in terms of events happening in order. If you think about a menu-based program, where the user is offered a list of options over and over, which may lead to actions and/or other lists of options, and so on, procedural programming describes this very well.

However, if you think about a modern word processor, operating system, or open-world game, describing these in terms of a series of steps is much more difficult. The user could choose to use any part of the program at any time in any order, and most of the parts interact with one another. It would be easier to think of these programs as groups of interacting components.

The *Object Oriented Paradigm* in programming design thinks of a program as a group of types of components, each of which has characteristics and behaviors. The types of components will be represented in the program by types of object, in Java called *classes*. The characteristics of the components will be represented by variables stored in the classes, and the behaviors by methods stored in the classes.

Remember that we want our design approach to make our lives easier. By having the code divided into classes in a way that follows our conceptual understanding of the situation our program reflects, we remove a lot of mental burden – the code reflects our thinking. Object orientation supports laziness in other ways too. Classes make our code modular – it is possible to pull a class out of one program and put it into another program that needs a similar type of component. We will also see that the standard ways we write classes in Java will help make it easier to update and modify our code when we talk about Encapsulation later in the course.

Classes

A class represents a type of component involved in a program. When designing a program using object orientation, the first step is to identify what the classes should be. Once classes are created, each class will be a valid type for variables in the program, just as good as built-in types like int or double.

Each class should be named with a meaningful name. A programming standard agreed on by most Java programmers is to give them names that start with a capital letter so that they stand out from keywords, methods, and variables. The class name should generally be singular – the class can be used to create many variables of that type, but each variable will be singular.

So, if our program will involve dogs and cats, we might create a class Dog and a class Cat, and once created, those become valid types for variables in that program. The values of these variables, based on the class, are called *objects* also known as *instances of the class*.

In Java, the keywords for starting a class are `public class`, and like other structures, we use curly braces { } at the beginning and end. Remember that each class in Java needs to live in a file with the same name as the class followed by .java. For the purposes of examples I may show the contents of several .java files all together like this:

```
 // in Dog.java
 public class Dog {
    // we will put more here soon
 }

 // in Cat.java
 public class Cat {
    // here too
 }

 // in MainClass.java
 public class MainClass {
    public static void main(String[] args){
       int x; // x is a variable of type int
       Dog fluffy; // fluffy is a variable of type Dog
       Cat muffy; // muffy is a variable of type Cat
       Dog d2; // d2 is another variable of type Dog
       Cat c2; // c2 is another variable of type Cat
    }
}
```

In the code above, we could say that “fluffy holds a Dog object”, or that “fluffy contains an instance of the class Dog” but when being casual, we would often just say that “fluffy is a Dog” just as we would say that “x is an int”.

Notice that only the MainClass class has the main method. This is the usual structure of a Java program: there are many classes representing components of the situation, with one class with main to store the main program that uses the other classes as tools.

Reference Variables

In Java, classes define a valid type for variables, just as good as the built-in types like int and double, but types based on classes *are* different from those. Types like int, double, and boolean are *primitive types* while types based on classes are *reference types*. These are also known in other languages as *pointers*.

A primitive variable just has to store a single value, probably as a single binary pattern on a single line of memory. We will soon be adding to the structure of our classes, putting code to represent characteristics and behaviors inside, so a variable representing an object based on a class has to have a copy of that more complex structure, which could contain many instance variables plus the many instructions to do behaviors. And in most situations, the different component types will interact, which may mean we need to have a variable of one reference type inside the class of another reference type. There might also be cases where an object of a component type needs to know about others of the same type.

For example, a dog might need to know about cats, and about other dogs. But having a whole Cat object nested inside a Dog object starts to be messy, and as soon as we try to nest a Dog object inside another Dog object, the computer would have to create another Dog object inside that, which has another Dog object inside that, which has another Dog object inside that, which has another Dog object inside that, which has another Dog object inside that … This is infinite recursion hell. We can never stop making Dogs[1]!

Instead of having Dog structures nested inside each other, we will give a Dog object a different way to access another Dog object: we will give it the ability to store the address in main memory where the other Dog is located. A variable that stores the memory address where data is stored instead of storing that data itself is called a pointer or reference.

[1] At first, infinite puppy sounds good, but remember, we never finish building them, so we can never pet these poor infinitely recursive puppers.

Remember that the main memory (RAM) of the computer is just a big table of numbered rows, and on each row we can store some data. The row numbers are called memory addresses. Since the rows are numbered, one kind of data we can store on a row is the memory address of another row in memory.

A variable is actually just a name for a particular row of memory anyway. Data is always stored at a memory address, variables just give us the convenience of using a memorable name instead of remembering a number. So a reference variable is also a name for a row of memory, but instead of storing the value directly there, we are using that row to store the number of another row.

This means that in Java, the variable and the actual structure of the object are separate things. The variable holds the memory address of the object. This separation will be *very important* when we get to more sophisticated programming tools like polymorphism.

Creating objects with new

When we do assignment for primitive types, we can just write the literal value we want to put in the variable, like 4.9 or “hello”. But we can’t do this for reference types because these include that whole class structure. Instead, we use the keyword new to create the structure in memory.

```
Dog d; // d is a dog variable, not yet holding any value
d = new Dog(); // create a Dog structure in memory
               // and store its address in d.
```

Notice that after the keyword new we used the name of the type with parentheses at the end[2].

In Java, programmers do not have direct access to the memory addresses our programs use; we cannot decide where in memory Java will store our objects. The values I use for these addresses in examples I will make up arbitrarily.

Let’s think about what is happening in memory when we do this. At the left I show some code in a main, at the right, the final state of part of memory (starting at address 100) after this code is done running, with previous values it had during the run of the program shown struck out. In all such examples, I will use MM in front of stored memory addresses.

```
// primitive variables
int x; // variable in memory
```

variable	*MM*	*Value*
x	100	~~3~~

[2] If you remember what parentheses after a name mean, you already suspect that what we are really doing here is calling a method to set up our Dog structure. For right now, this is a default version of a method called a *constructor* that we will learn to write later.

```
       // on line 100
int y; // variable at MM101

// variables that can store memory
// addresses of Dog objects
Dog dogA; // variable at MM102
Dog dogB; // variable at MM103

// give primitives values
x = 3;
y = 5;

// create new Dog objects
// and store their addresses
// in the variables
dogA = new Dog(); // MM200
dogB = new Dog(); // MM300

// copy value from variable
// then change original
y = x;
x = 99;

// copy value from variable
// then change original
dogB = dogA; // copy value MM200
dogA = new Dog(); // MM400
```

		99
y	101	~~5~~ 3
dogA	102	~~MM200~~ MM400
dogB	103	~~MM300~~ MM200
	200	*Copy of structure of the Dog class*
	…	
	300	*Copy of structure of the Dog class*
	…	
	400	*Copy of structure of the Dog class* …
	…	

Notice that the reference variables really behave just like the primitives, they each can hold a value, which can be copied and replaced. It is just that now this value is a location in memory. Two reference variables can hold the same memory address value, in which case we would say that they both *point to* the same object.

Instance Variables

For each class type in our program, there may be various characteristics we need to keep track of. Some of these may be permanent characteristics, such as the name or breed of a dog, others may be used for tracking a current state that can change over the run of the program, such as whether the dog is hungry or asleep. As usual, we will store these values in variables, but these variables will be collected inside the class.

These are known as *instance variables*, *fields*, or *attributes* of the class. To add an instance variable to a class we simply declare it as a variable inside the body of the class structure rather than inside a method in the class.

To declare a variable we always put first the type, then the name. For instance variables we also put the *access modifier* public in front of the type (we will later see that public is a bad idea and get a better alternative). We usually do not initialize instance variables when we declare them (we will get a better approach to giving them initial values later).

```
pubic class Dog {
   public String name; // instance variable for dog’s name
   public int age; // dog’s age
   public boolean awake; // dog currently awake?
}
```

When we have a variable of a class type, we can access the instance variables in that object using the *dot operator* syntax. A period goes between the name of the class-type variable and the name of one of that class’ instance variables. We usually pronounce this dot operator as a possessive s so `fluffy.age` is pronounced “fluffy’s age”.

Values that live in variables inside an object are still values of their type and can be used anywhere else we would use such values. We can print them; we can use them to do math if they are numbers; we can (later) use them to make decisions if they are booleans.

Each instance of the class has its own copy of the full structure of the class, including all the instance variables. So, assuming the Dog class just defined, if we are in main:

```
int x;
Dog fluffy = new Dog(); // whole Dog structure
Dog d2 = new Dog(); // whole Dog again

// use the dot operator to access fluffy’s name
fluffy.name = “Fluffywuffykins”;

name = “Wrong”; // error, we didn’t create a variable name
      // we can only access the name in the dog class
      // by using the dot operator for a specific dog
```

```
int age = 5; // legal, but this age is unrelated to
                // the dogs’ ages

// d2’s name is a separate variable with its own value
d2.name = “Mr Dog”;

// fluffy’s age
fluffy.age = 2;

// fluffy.age is an int, we can do math with it:
x = fluffy.age * 3; // x is now 6
d2.age = fluffy.age + x; // d2’s age is now 8

// d2 is awake
d2.awake = true;
```

We now have two different kinds of variables in Java. The kind of variables we are used to creating inside a method, like x, d2, and fluffy in the example above, are *local variables,* but we now also have instance variables.

When we started having local variables, we saw it was a syntax error to use a local variable before assigning it a value, because local variables do not have a value until assigned. Instance variables do not work like this. Instead, instance variables start off having default values determined by their types:

- 0 for numbers
- false for booleans
- null for all other types.

Now that we have a structure for Dog, lets think about memory again:

```
// create a dog variable
// at MM100, no value yet
Dog dogA;

// create a dog object and store
// its address in the variable
```

variable	*MM*	*Value*
dogA	100	~~MM200~~ MM400
dogB	101	~~MM300~~ MM200
	200	name: ~~null~~ ~~“Abe”~~

```
dogA = new Dog();
   // MM100 now storing MM200
Dog dogB = new Dog();
  // MM101 storing MM300

// currently different objects
dogA.name = “Abe”;// affects MM200
dogB.age = 3; // affects MM300

// make them the same object
dogB = dogA; // now both are MM200
dogB.name = “Bob”; // affects MM200

// change A to different object
dogA = new Dog(); // MM400
dogA.name = “Ann”; // affects MM400
dogA.awake = true; // affects MM400
```

	“Bob” age: 0 awake: false
…	
300	name: null age: ~~0~~ 3 awake: false
…	
400	name: ~~null~~ “Ann” age: 0 awake: ~~false~~ true
…	

Now we can start to understand what it means that a reference variable holds the memory address of an object and that the variable and object are separate:

It means that a variable can be associated with different objects during the run of the program; for a while dogA was associated with the object at MM200, but later the same variable was associated with the object at MM400. So although the lines `dogA.name = “Abe”;` and `dogA.name = “Ann”;` look like they’re talking about the same variable, they affect totally different objects in memory.

It also means that two variables can be associated with the same object at the same time; for a while in the example dogA and dogB were both holding the address MM200, so the line of code `dogB.name = “Bob”;` could be re-written `dogA.name = “Bob”;` at that point in the program and the code would do exactly the same thing.

You can think of a reference variable as being like a remote control that can be paired with an existing piece of electronics, or like a leash that can control an animal. But the leash isn’t the animal itself, just a way of controlling it.

Changing dogA from one object to another is like hooking a leash first onto one dog’s collar, and later onto another dog’s collar. When dogA and dogB had the same value, it was like hooking two leashes to the same dog’s collar.

Object Instance Variables

By creating classes, we have added to the possibilities for variable types in our programs, and that applies to instance variables! Since a class defines a type that is valid for variables, we can put an instance variable of one class type inside another class.

Let’s have one class representing cats, and a second representing dogs, who can be friends with cats:

```
 public class Cat {
    public String name; // cat’s name
    public double weight; // cat’s weight
 }

 public class Dog {
    public String name; // dog’s name
    public Cat buddy; // dog’s buddy, which is a Cat
 }
```

Now in a main we can say:

```
    // create dogs
    Dog fluffy = new Dog();
    fluffy.name = “Fluffers”;
    Dog d2 = new Dog();
    d2.name = “Spot”;

    // create a cat, who is fluffy’s buddy
    fluffy.buddy = new Cat();

    //fluffy’s buddy is named “Mr. Whiskers” and
    // weighs 4.2 lbs
    fluffy.buddy.name = “Mr. Whiskers”;
    fluffy.buddy.weight = 4.2;

    // a separate cat unrelated to the dogs
    Cat ginger = new Cat();
    ginger.name = “Ginger McBeans III”;
```

Notice that the instance variables of a class are local to that class. The `name` instance variable inside Cat does not interfere with or create a special relationship with the `name` instance variable inside Dog, and neither can be directly accessed outside the class without using the dot operator to explicitly say which Cat or Dog it is related to

We never set up a Cat object for d2’s buddy variable, so that variable still has the default value null. Null is used to mean no memory address. Seeing that d2’s buddy is null, we would interpret that to mean that d2 doesn’t have a cat that it is buddies with. Having the variable means that, like all dogs, it has the capability to be buddies with a cat, but there is no cat it is currently friends with. If we later wanted to represent that Spot has become friends with Ginger McBeans III, we could say:

```
d2.buddy = ginger;
```

Which means we would again have two variables pointing to the same object and in that situation these two lines would do exactly the same thing:

```
ginger.weight = 9.9;
d2.buddy.weight=9.9;
```

Having dogs that are friends with cats is nice, but what about dogs being friends with other dogs? This is no harder:

```
// dog that can have another dog
// as a best friend
public class Dog {
   public String name;
   public int age;
   public Dog  bestFriend;
}
```

Let’s use this in a main and see what happens under the hood:

```
Dog fluffy = new Dog();
fluffy.name = “Fluffy”;
fluffy.age = 3;
```

variable	*MM*	*value*
fluffy	100	MM200
spot	101	MM300

```
Dog spot = new Dog();
spot.name = “Spotty”;
spot.age = 2;

// they become each
// other’s best friend[3]
fluffy.bestFriend = spot;
spot.bestFriend = fluffy;

System.out.println(spot.name  +
     ” is best friends with “ +
     spot.bestfriend.name +
     “ whose best friend is “ +
   spot.bestfriend.bestfriend.name”);
```

fluffy	200	name: ~~null~~ “Fluffy” age: ~~0~~ 3 bestFriend ~~null~~ MM300 …
	…	
spot	300	name: ~~null~~ “Spotty” age: ~~0~~ 2 bestFriend: ~~null~~ MM200 …
	…	

The last line would print “Spotty is best friends with Fluffy whose best friend is Spotty”. Since the bestFriend variable is a Dog, and every Dog has a bestFriend variable, and we have made these two Dogs each other’s bestFriend, we could go around this circle indefinitely. Make sure you understand why this would be a perfectly valid thing to write at the end of the above code:

```
fluffy.bestFriend.bestFriend.bestFriend.bestFriend.bestFriend.bestFriend.bestFriend.bestFriend.bestFriend.bestFriend.bestFriend.bestFriend.bestFriend.bestFriend.bestFriend.bestFriend.bestFriend.bestFriend.bestFriend.bestFriend.bestFriend.bestFriend.bestFriend.bestFriend.bestFriend.bestFriend.bestFriend.bestFriend.bestFriend.bestFriend.bestFriend.bestFriend.bestFriend.bestFriend.bestFriend.bestFriend.bestFriend.bestFriend.bestFriend.bestFriend.bestFrie
```

[3] Becoming each other’s best friend requires BOTH lines, fluffy has to know spot’s address AND spot has to know fluffy’s address.

```
nd.bestFriend.bestFriend.bestFriend.bestFriend.bestFriend.bestFri
end.bestFriend.age = 8;
```

Class Methods

An object is not just a user-defined data type that combines other data types, it also allows us to group together behaviors specific to the type of component by putting methods inside the class to represent behaviors that are being done by an instance of the class.

For now, we will only be writing very simple methods that all have the same format for behaviors. They will have the keyword public and the type void before the name, and empty parentheses after the name. Just like choosing names for variables, think about choosing names for methods that will make your code easy to read.

Any code you can write inside the main method, you can also write in these behavior methods, including including declaring variables Each method in the class also has access to all the class' instance variables (no dot operator needed).

```
 public class Dog {
    public String name; // instance variable for the Dog’s name
    public int age; // the dog’s age
    public boolean awake; // is the dog currently awake?

    // dogs can bark
    public void bark() {
          System.out.println(name + “ says woof”);
    }

    // dogs can sleep
    public void sleep() {
          System.out.println(name + “ goes to sleep”;
          awake = false;
    }

    // dogs can do math
    public int convertAge() {
          int peopleYears = age * 7
          return peopleYears
    }
```

```
}
```

We can then access an object’s behavior methods using the dot operator in a main (or any other method).

```
Dog fluffy = new Dog();
fluffy.name = “Fluffers”;
fluffy.age = 2;

Dog d2 = new Dog();
d2.name = “Spot”;
d2.age = 1;
d2.awake = true;

// fluffy does fluffy’s bark behavior
fluffy.bark(); //print “Fluffers says woof”
// d2 does d2’s bark behavior
d2.bark(); //print “Spot says woof”

d2.sleep(); // print “Spot goes to sleep”
    // d2.awake becomes false

System.out.println(“in people years, “ + d2.name + “ is “ +
    d2.convertAge() + “ years old”);
```

Note that we wrote the behavior methods to be general, to work for any dogs we ever create, by using the name and age variables, not specific values that were later used in main. Even if we think right now that a class will only be used once for a specific variable with specific values in a specific main, we should still write its behavior methods to work generally.

We have to be able to write a method in a class to use its instance variables without knowing what the values of those instance variables will be. We should generally assume that there will be multiple instances of the class created, and each instance of the class should be able to do its behaviors and maybe get different results depending on the current values of the instance variables.

Also note that main uses the parts of the class – its instance vars and methods – by creating instances of that class. But the class doesn't use the variables from main. Creating the class is like setting up a toolbox. The main then uses those tools. Not the other way around.

The behavior methods defined in a class only run if they are called by the program starting from main. In the program above, we never told fluffy to sleep, so the sleep method never ran for that dog. If we had not called the sleep method for either of them, the sleep method never would have run at all.

During the design phase of programming, we decide what characteristics and behaviors each class type should have, and then we give it instance variables and methods for those. When writing the main program we may not use all the characteristics and behaviors. That *could* mean that our original design was wrong, including things that don't make sense for the program we were designing after all. But it could also mean that we were future-proofing ourselves, because although the current version of the program doesn't need those tools, we can forsee that such tools will later be needed as the program changes. Also, in many cases we re-use classes across programs – this re-use saves time and work, that's a major benefit of object orientation! – and the tools one program doesn't need could be needed in a different program.

toString

There is one more method that every class in Java should have unless it is just the class that holds the main method. This special method is called toString. Remember that when we print or concatenate other types onto a String, those types are implicitly cast to Strings so that they take readable form. For numbers and Booleans, it's pretty clear how they should print, but what should we see when we print a Dog or a Cat?

That's a design decision for the creator of the class! What an object base on your class looks like when it is printed is defined in the toString method – it is the method that tells Java how to cast your class *to* a *String*.

toString looks a little different from our simplified behavior methods, but the basic structure of all toString methods must look the same.

```
public String toString() {
  return String representation of your class goes here ;
}
```

If you don’t write a toString for your class, when you print objects of your class type you will see something that looks like the name of your class and some garbage (actually Java’s identifier for that object) that isn’t useful to humans.

If your class has instance variables, it should definitely have a toString, and usually the toString should include their values, so that you can easily see as much information as possible when you print. We create this String representation by just concatenating the instance variables with some spacing and text to clarify what we’re seeing.

Here is a simple toString for a simple class:

```
public class Cat {
   public String name; // cat’s name
   public double weight; // cat’s weight

   public String toString() {
         return “the cat “ + name + “ (“
              + weight + “lbs)”;
   }
}
```

When we have more complex classes, we may have a harder time including instance vars in the toString. In the Dog class here, I’m showing some different options for toString; a real class can only have one toString method!

```
public class Dog {
   public String name; // dog’s name
   public Dog bestFriend; // dog’s best friend, a Dog
   public Cat buddy; // dog’s buddy, a Cat
   public boolean awake;

   // option 1
   public String toString() {
         return name + “ the dog (awake? “ + awake + “)”;
   }
   // option 2
   public String toString() {
         return name + “ the dog (awake? “ + awake
              + “) has bestFriend ” + bestFriend.name
              + “ and buddy “ + buddy.name;
   }
   // option 3
   public String toString() {
         return name + “ the dog (awake? “ + awake
              + “) has bestFriend ” +
              + “ and buddy “ + buddy;
   }

}
```

The first option above shows just using primitive instance vars and ignoring the reference variables. This might print something like “the dog Fluffy (awake? false)” but it leaves out the best friend and the buddy. There will be more elegant ways to deal with booleans later, but for now we can just work around the fact that they print as “true” or “false”.

The second option would include the best friend’s name and the buddy’s name in ideal cases. But if the dog doesn’t have a best friend or doesn’t have a buddy, then trying to access this name causes an exception, a runtime error that blows up your program (see the section later on Null Pointers). We need conditionals to be able to check for situations like that before we’ll be able to have a toString like this while avoiding that problem.

The third option uses the whole Dog bestFriend and the whole Cat buddy instead of trying to pull out the name. So we are putting a Dog and a Cat in the middle of a String with concatenation. If they are null, this just includes literally “null” which isn’t pretty, but doesn’t cause an error. Otherwise this means Java has to cast the Dog and the Cat to Strings, which triggers a call to toString. For the buddy, that works fine, we are just including the Cat’s toString in the Dog’s toString. But for the bestFriend this means the Dog’s toString is triggering the Dog’s toString which triggers the Dog’s toString which triggers the Dog’s toString… Infinite recursion hell, which eventually breaks our program with an error. So we can do it this way for variables of other class type, but not for our own.

For right now, just make sure you are including all primitive (and String) instance variables in your toString.

Ideally, your toString’s String representation of the class should be something that could be dropped into the middle of a sentence, so that if we had a Dog d1 and a cat c1, we could get sensible results from printing things like

```
System.out.println("Today I met " + d1 + " at the dog park and
   then went home to feed " + c1  + "his tuna treats."
```

But in some cases it is more useful to make a multi-line toString that shows lots of information but really needs to be printed by itself.

Null Pointers

The value null represents an invalid memory address. Trying to use the dot operator on a reference variable that is holding the value null is a runtime error in java, which is called a *null pointer exception*. This stops the program immediately. It makes sense that the program can't run in this case because null means there is no object that holds the instance variable or has the method we are trying to use.

A null pointer exception actually is the best thing that could happen if we have not initialized the value of a reference; Java defaulting to nulls is actually saving us a lot of headaches. Other languages let variables use "garbage values" – whatever binary pattern happened to be sitting the row in memory the variable represents (data from some previous program perhaps). Suppose the variable is a reference and that garbage value is a valid value for a memory address. If we try to write data to something at that value, we could end up overwriting any other data *or instruction* anywhere in Main Memory![4] That kind of error might mean the program looks like it is running fine, but some completely different program or other part of the program suddenly breaks for no apparent reason; this is very hard to debug! Stopping the program immediately and saying where the error is (the exception should show you a line number!) is incredibly convenient.

A null pointer exception is one of the most common types of error when programmers are learning to write code with reference types, because it is easy at first to forget which instance variables have not been initialized yet.

Let's have cats and dogs that know about both cats and dogs:

```
public class Cat {
   public String name; // cat's name
   public double weight; // cat's weight
}
```

[4] In modern systems under virtual memory, our program should never be allowed to write to a memory address outside the pages of main memory that the operating system assigned to our program. This limits a garbage value for a pointer to only wreaking unspeakable havoc on our program's own instructions and data. Yay?

```
public class Dog {
   public String name; // dog’s name
   public Cat buddy; // dog’s buddy, a Cat
   public Dog bestFriend; // dog’s best friend, a Dog
}
```

Now in a main let’s say:

```
Dog fluffy = new Dog();
fluffy.name = “Fluffy”;

Dog spot = new Dog();
spot.name = “Spotty”;

// error! bestFriend is null
fluffy.bestFriend.age = 4;

// error! buddy is null
fluffy.buddy.name = “Ginger”;

// make this not null
spot.bestFriend = fluffy;
spot.bestFriend.age = 3;

// but this is still null
spot.bestFriend.bestFriend.age = 1;
```

variable	*MM*	*value*
fluffy	100	name: ~~null~~ “Fluffy” age: ~~0~~ 3 buddy: null bestFriend: null …
	…	
spot	200	name: ~~null~~ “Spot” age: 0 buddy: null bestFriend: ~~null~~ MM100 …
	…	
	…	

Note that since we set spot’s bestFriend variable to point to fluffy, it stopped being null, but that did not automatically make fluffy’s bestFriend variable point to spot (not all friendships are requited, sad to say).

Memory Management

In a previous example, we created a Dog using new, but then changed the variable that held that dog’s address to point to a different Dog, which meant we lost access to the first Dog object. The object still existed in main memory, but our program no longer had access to it.

In some languages, it is the responsibility of the programmer to manage how their program uses main memory space. If we use the `new` operator to create an object and then lose access to it, in these languages we can now no longer reuse this memory space to store something else – it is still marked as being in use – but we cannot actually use the data there. This is called a *memory leak*.

In these languages, the programmer must use another operator called `delete` to explicitly give back the space for an object so that space can be re-used. They have to think carefully about when to do this. Too early would mean destroying data while we still need it, too late could lead to running out of memory space.

We're glad we work in Java where we don't have to deal with that!

In Java, the program's memory space automatically undergoes a process called garbage collection where objects are checked to see whether they are still reachable and if not, be automatically marked as space to be re-used.

Garbage collection is much more convenient, but we can't control when garbage collection happens, so it could potentially happen that in a program we create a lot of objects at a point when garbage collection hasn't run for a while, and our program would suddenly slow down as garbage collection runs and tries to keep up. In modern versions of Java though, a little bit of garbage collection happens periodically throughout a program's run to try to keep the speed level.